# Machine Fundamentals

Professor Elizabeth Scott

Department of Computer Science
Egham, Surrey TW20 0EX, England

These notes accompany the first year course CS1870. They provide a basic cover of the course material. You should supplement them by taking notes in lectures and by reading relevant text books.

# Contents

# Chapter 1

# Introduction

Mathematics is intimately involved with computer science in two basic ways.

- ◇ Ultimately, computers perform mathematical calculations. This is historically why computing was developed, (by mathematicians such as Alan Turing).
- ◇ Many of the major uses for computers are essentially solving mathematical problems. Physicists have used computers for a long time to predict results of experiments using mathematical models, and to analyse data taken during experiments.

Whatever the application, computers ultimately execute algorithms which have been defined mathematically. The algorithms are translated in to a series of elementary operations which the computer then performs. In CS1860 you studied aspects of mathematics which underpin its use as a language for defining algorithms. In this course we focus mostly on the first point above, studying aspects of mathematics which are fundamental to the behaviour of computing machines, both 'real' computers and idealised machines such as finite state automata.

Essentially all computers do is read and write binary sequences from memory locations and perform simple operations such as addition and logic computations. The set of available arithmetic operations is usually very small, and the operations are usually primitive. The combination and interpretation of sequences of these operations is what allows modern computers to achieve enormous computational power and complexity.

In this course we study the representation of numbers and numeric operations, logic and its role in computation, idealised machines such as finite state automata, elements of computer hardware and low-level (assembler) programming.

One of the main skills that a computer scientist needs is to be able determine how to solve a problem in a way that allows a program to do it to be written. A computer has no ability to 'think', it only 'knows' what it has been told and it can only execute actions that it has been given. For example, a compiler is a program which reads in a program written in some programming language, and outputs a program written in the machine's assembly language which has the same meaning. The material in this course underpins many parts of compiler design and generation.

## 1.1 Aims

By the end of this course the student should be able to:

- ◇ Understand and be able to use binary representations of numbers in a computer.

- ◇ Have an understanding of elementary logic and normal forms.
- ◇ Be able to construct logic circuits and switching circuits for logical expressions.
- ◇ Have an understanding of MIPS assembly language.
- ◇ Be able to write regular expressions to describe sets and construct finite state automata to recognise regular sets.
- ◇ Understand the construction and use of push down automata.
- ◇ Have some understanding of the absolute limitations of computers.

## 1.2 Course outline

Binary representations of numbers. Two's complement representation. Floating point representation.
Propositions, logical connectives, truth tables. DNF and CNF. Logical equivalences.
Predicate logic. Logical inference and proof.
AND, OR, NOT, NAND gates. Logic circuits. DNF as sum of minterms. Karnaugh maps.
Full and half adder, ripple carry adder.
N and P type transistors. Cannonical pull-up and pull-down circuits.
MIPS assembly language and SPIM.
Regular expressions, finite state automata, lexical analysis.
Push down automata, Turing machines and the Chomsky hierarchy.
Non-computability.

## 1.3 Books

These lecture notes underpin the course. Print them out and bring them to lectures. Read them in advance and annotate them. Also use the Web to find related material.

Towards the end of the course there will be a MIPS lab session with a work sheet. The Web is a good place to read more about MIPS.

There is no set book for this course, and there is no book which alone covers the material presented in the course. There are two basic texts which are useful.

1. Kenneth Rosen, *Discrete Mathematics And Its Applications*, 6th Edition, McGraw-Hill, 2007. ISBN 0-07-113974-5. RHUL Library: 512.23 ROS

2. J. Glenn Brookshear, *Computer Science: An Overview*, Addison-Wesley, many editions, last (tenth) 2008. RHUL Library: 001.64 BRO

Other books that you might find useful.

3. Daniel Cohen, *Introduction To Computer Theory*, second edition, Wiley, 1991. ISBN 0-471-13772-3.

4. Peter Linz, *Introduction To Formal Languages And Automata*, third edition, Jones and Bartlett, 2001. ISBN 0-7637-1422-4.

5. John Hopcroft and Jefferey Ullman, *Introduction To Automata Theory, Languages And Computation*, Addison Wesley, 1979. ISBN 0-201-02988-X.

## Course organisation

Three lectures per week.

Five advisee tutorials. Each tutorial has an associated compulsory exercise sheet.

The three coursework assignments will count for 10% (split evenly between the three assignments);

Provisional Examination Rubric:
Answer ALL questions
Time Allowed: $1\frac{1}{2}$ hours
Calculators NOT Permitted

**Attendance requirements**

- ◇ You are required to attend all lectures and advisor tutorials. This is a College requirement and we take a register.
- ◇ You are required to do all the set assignments and exercise sheets.

# Chapter 2

# Numbers and computers

First we consider how can we represent numbers inside a computer.

## 2.1 Using bases

Our number systems tend to be *positional* with respect to some base. Normally the base 10 system is used, so 137 represents

$$1 \times 10^2 \ + \ 3 \times 10^1 \ + \ 7 \times 10^0$$

Representing numbers this way in a computer is not common, although not impossible. The 19th century computer designed by Charles Babbage had columns of cog wheels, with the digits 0 to 9 etched on them. Then each column contains a number represented using the decimal interpretation, each cog on the column corresponding to a power of 10. You can see a version of Babbages machine being described and operated at the following site
`http://www.youtube.com/watch?v=0anIyVGeWOI`
It shows the cogs and how precisely they needed to be made.

Base 10 is not the only base that has been adopted. The Ancient Babylonians (1750 BC) used a sexadecimal system, that is base 60. This is why one hour is divided into 60 minutes and 1 minute into 60 seconds. The advantage of this system is that 60 has lots of divisors but a disadvantage is that we need to design and remember 60 different symbols, and the multiplication tables would be large.

Many computer languages use hexadecimal, base 16, with symbols 0,. . . ,9,A,B,C,D,E,F.

Electrical computers lend themselves to a binary, base 2, system because 0 and 1 can be represented by 'on' and 'off'. Historically this has been done via relays, vacuum tubes, transistors and voltages on integrated circuits. You can see vacuum tube memory on the Bletchley Park rebuild of the Colossus machine
`http://en.wikipedia.org/wiki/File:ColossusRebuild\_11.jpg`
`http://en.wikipedia.org/wiki/File:Colossus.jpg`

The numbers 0 and 1 are called the *binary* digits, and may be referred to as *bits*. The term bit was introduced by John Tukey, an American statistician and computer scientist, in 1946. (Binary would have been much more expensive for Babbage, needing many more cogs and columns.)

There are many methods that have been used to implement bits. In magnetic storage devices (Hard Rigid Disk, Floppy, Zip, Tape, etc.) magnetized areas of the media are used to represent binary numbers: a magnetized area stands for 1, and the absence of magnetization means 0. Flip-flops are electronic devices that can only carry two distinct voltages at their outputs, traditionally 0 and 5 volts. They can be switched from one state

to the other state by an impulse. Optical and magneto-optical storage devices use two distinct levels of light reflectance or polarization to represent 0 or 1.

The ideas and results covered in this course apply to all methods of representing bits, we abstract away from the hardware details.

## 2.2 Binary numbers

### 2.2.1 Positional representations

Recall that *decimal numbers* are written positionly, using coefficients of powers of 10.

$$137 = (1 \times 100) + (3 \times 10) + (1 \times 7)$$

Similarly, *binary numbers* are sequences of 0s and 1s, interpreted as coefficients of powers of 2, with the highest power to the left.

$$11010 \;=\; (1 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) \;=\; 26$$

It is usually easiest when converting long binary numbers to decimal number to start from the right rather than the left.

$$11010 \;=\; (0 \times 2^0) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^3) + (1 \times 2^4) \;=\; 26$$

The value of the expression is the same, the left-most binary digit still corresponds to the highest power of 2, we have just written the powers out in the opposite order.

Of course decimal 11 (that is base 10) is a different value to binary 11 (base 2), which is decimal 3. So it is important to know the base. If there is any possibility of confusion we write the base as a subscript. So $11_2$ is binary 11, which has decimal value 3.

We carry this notation over to equality statements, so we write

$$11_2 \;=\; 3 \qquad \text{or even} \qquad 11_2 \;=\; 3_{10}$$

Another example:

$$11101 \;=\; (1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0) \;=\; 29$$

$$11101_2 \;=\; 29 \qquad \text{or} \qquad 11101_2 \;=\; 29_{10}$$

**Exercise** Write (i) $10010_2$ and (ii) $110010_2$ in decimal form.

Here are the first 32 binary integers.

```
 0:   0        8:   1000     16:   10000     24:   11000
 1:   1        9:   1001     17:   10001     25:   11001
 2:   10      10:   1010     18:   10010     26:   11010
 3:   11      11:   1011     19:   10011     27:   11011
 4:   100     12:   1100     20:   10100     28:   11100
 5:   101     13:   1101     21:   10101     29:   11101
 6:   110     14:   1110     22:   10110     30:   11110
 7:   111     15:   1111     23:   10111     31:   11111
```

### 2.2.2 Padding with leading zeros

It does make sense to write decimal numbers in the form, say, 004502, with zeros to the left. However, this is rarely done. There are some exceptions though, 007 is very well known and hotel rooms are sometimes number 01, 02 etc, if they are on the ground floor.

The opposite is true for binary numbers. When used in hardware, binary integers tend all to have the same length, achieved by packing with zeros from the left.

```
 0:   00000     8:   01000     16:   10000     24:   11000
 1:   00001     9:   01001     17:   10001     25:   11001
 2:   00010    10:   01010     18:   10010     26:   11010
 3:   00011    11:   01011     19:   10011     27:   11011
 4:   00100    12:   01100     20:   10100     28:   11100
 5:   00101    13:   01101     21:   10101     29:   11101
 6:   00110    14:   01110     22:   10110     30:   11110
 7:   00111    15:   01111     23:   10111     31:   11111
```

Notice, even numbers end with 0 (on the right), odd numbers end with 1.

### 2.2.3 Finding binary representations

Given a binary number we can find the equivalent decimal representation by adding together the corresponding powers of 2.

Similarly, given a decimal number we can find the equivalent binary representation by subtracting powers of 2.

Start by finding the highest power of 2 which is less than or equal to the binary number. Then subtract this power of 2 from the number to get the remainder. For example, we convert $137_{10}$ to binary as follows.

$$2^7 = 128 < 137 < 256 = 2^8 \quad \textit{remainder} \quad 137 - 128 = 9$$

Repeat this process on the remainder,

$$2^3 = 8 < 9 < 16 = 2^4 \quad \textit{remainder} \quad 9 - 8 = 1 = 2^0$$

Carry on in this way until the remainder is a power of 2.

$$\begin{aligned} 137 &= 128 + 9 \;=\; 128 \;+\; 8 \;+\; 1 \\ &= (1 \times 128) + (0 \times 64) + (0 \times 32) + (0 \times 16) + (1 \times 8) + (0 \times 4) + (0 \times 2) + (1 \times 1) \\ &= 10001001_2 \end{aligned}$$

Another example:

$$\begin{aligned} 1000 &= 512 \;+\; 488 \;\;=\;\; 512 \;+\; 256 \;+\; 232 \;\;=\;\; 512 \;+\; 256 \;+\; 128 \;+\; 104 \\ &= 512 \;+\; 256 \;+\; 128 \;+\; 64 \;+\; 40 \;\;=\;\; 512 \;+\; 256 \;+\; 128 \;+\; 64 \;+\; 32 \;+\; 8 \\ &= 1111101000_2 \end{aligned}$$

**Exercise** Write (i) 97 and (ii) 232 in binary form.

### 2.2.4 Using binary numbers - counting

Binary numbers can be used like decimal ones.

- ◇ Counting in decimal:
  0, 1, 2,. . . , 9, 10, 11,. . . , 19, 20, 21,. . . , 99, 100, 101,. . .
- ◇ Counting in binary:
  0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100,. . .

In both cases it is the same rule: as a digit reaches its maximum allowable value, it becomes 0, and the digit to its left is incremented by 1 (recursively).

**Exercise** Write down the next four numbers in the binary series.

### 2.2.5 Using binary numbers - arithmetic

Binary numbers add and multiply together in the same way as decimal numbers. We have

$$2^0 + 2^0 = 1 + 1 = 2^1 \qquad \text{so} \qquad 1_2 + 1_2 = 10_2$$

$$3 = 2 + 1 = 2^1 + 2^0 \qquad \text{so} \qquad 10_2 + 1_2 = 11_2$$

$$5 = 2 + 3 = 2^1 + 2^1 + 2^0 \qquad \text{so} \qquad 10_2 + 11_2 = 101_2$$

In general we add and multiply binary numbers using position and carry from the left, as for decimal numbers.

```
                                                    101
                                                  ------
 1000110          1111001          1011        101 | 11001
+ 110011        - 110011         x 101             -101
 -------          -------          ----              ---
 1111001          1000110          1011                101
                                  101100               -101
                                  ------               ---
                                  110111                 0
```

The following are the addition and multiplication table for the binary digits.

Addition table:

| + | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 10 |

Multiplication table:

| × | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

**Exercises** Carry out the following operations:

- ◇ $10111001_2 + 11001101_2$
- ◇ $11011100_2 - 1110111_2$
- ◇ $1110_2 \times 1101_2$
- ◇ $111001_2/101_2$ (with remainder)

### 2.2.6 Binary is not the only way

It is natural, as we have discussed, to interpret a string of numbers positionly and compute its value relative to some base. So $1101100_2$ is thought of as representing sums of powers of 2. However, this is not the only possibility, even for electronic machines built of cells which have just two states ON/OFF.

An alternative representation, which was used by some IBM machines, is called the *2-of-5 code*. In this representation decimal digits are represented by binary sequences of length 5, exactly two of which are 1s.

There are several systems, for example in the IBM version

```
11000 is 1     10010 is 3     00110 is 5     01001 is 7     00011 is 9
10100 is 2     01010 is 4     10001 is 6     00101 is 8     01100 is 0
```

Using this, a binary string represents a decimal, base 10, number.

$$110001001001001 \quad = \quad 11000\ 10010\ 01001 \quad = \quad 137$$

Although modern computers don't use this representation, it has the advantage of *error detection*. In the binary representation, every string represents a number, so if there is data corruption this cannot be detected. If one of digits in a 2-of-5 code element is corrupted then the result will be a non-code element and the system will detect the error.

Binary systems can be made to detect one error by the introduction of a check bit, but this is less powerful than 2-of-5 detection which can identify which decimal digit contains the error. Of course the implementation of addition and multiplication using the 2-of-5 representation is more complicated than the add-and-carry method for positional representations.

## 2.3 Signed binary numbers

The numbers we have discussed so far are "unsigned integers": non-negative integer numbers. Variables of such types can be declared in C (and by extension in C# and C++) but not in Java. We need to be able to represent negative numbers, and indeed non-integer numbers such as fractions.

### 2.3.1 Bounded size

There is no upper bound on the size of an integer but computers are finite so there is a limit on the size of number they can represent. In practice the number of digits which is used to represent a number, the word size, is specified. If the word size is 32, numbers are 32 digits long. Modern general purpose computers are typically 32-bit or 64-bit, but 16-bit machines are used and embedded systems often have other word sizes. For the examples in this course we will use small word lengths to allow readable examples.

### 2.3.2 Words and bytes

As we have said, a computer's memory is usually divided into *words* typically 16, 32, or 64 bits. The largest number that a 64-bit word can hold using positional base-2 representation is $2^{64} - 1$.

Sometimes a word is holding several numbers, for example when the word is representing a machine instruction, and sometimes a number is represented using two or more words (allowing larger numbers to be represented on smaller word length machines).

A common sub-word size is 8 bits, called a *byte*. There are $2^8 = 256$ bytes.

```
00000000 is 0          11111110 is 254
00000001 is 1          11111111 is 255
```

The largest unsigned byte number is $255 = 2^8 - 1$.
It will often be convenient to talk about half-bytes (4 bits), e.g binary 1111 is decimal 15.

**Exercise** Write down all possible 4-bit strings and the unsigned integers they represent.

In Java we can explicitly specify certain types of integer representation, the default `int` uses 32 bits, `byte` only uses 8 bits, and short uses 16 bits, both of which save memory at the expense of restricting the size of integer that can be used. The data type `long` uses 64 bit integer representation, allowing larger values than are possible with `int` but with an efficiency impact in a 32 bit environment.

This is not a full specification of the integer data types, as we need to have negative as well as positive integers. For bytes this usually means having -8 to 7 rather than 0 to 15.

We now consider several possible representations which permit negative numbers.

### 2.3.3 Sign-magnitude representation

One way to represent negative numbers is to store the sign of a number as an additional piece of information. This corresponds to our usual mathematical notation, $+3$ and $-102$ etc. There are only two signs, `-` and `+` so one of the bits, usually the left-most, can be used for the sign. The remainder of the binary sequence then represents the absolute value of the number.

The usual convention is to use 0 for $+$ and 1 for $-$. For example, in an 4-bit representation, 3 is represented as 0011 and $-2$ is represented as 1010.

There are two particular disadvantages of the sign-magnitude representation:

- ◇ it is somewhat complicated to define arithmetic operations (the left-most bit plays a special role)

- ◇ there are two representations of 0 ($+0$ is represented as 0000 and $-0$ is represented as 1000)

### 2.3.4 Excess representation

The excess representation uses modular arithmetic interpretations of numbers. For example, -1 is 7 modulo 8. The fist bit is used to say whether the number is the actual number, when the first bit is 1, or its negative modular equivalent, when the first bit is 0.

For example, 1111 corresponds to 7 and 0111 corresponds to $-1$, which is the same as 7 modulo 8.

Here is a full list of the excess representation for 4-bit numbers. For the sequences beginning with 1 the interpretation is the same as for the signed bit representation.

| Bit pattern | Signed integer value (excess notation) |
|---|---|
| 0000 | $-8$ |
| 0001 | $-7$ |
| 0010 | $-6$ |
| 0011 | $-5$ |
| 0100 | $-4$ |
| 0101 | $-3$ |
| 0110 | $-2$ |
| 0111 | $-1$ |
| 1000 | 0 |
| 1001 | 1 |
| 1010 | 2 |
| 1011 | 3 |
| 1100 | 4 |
| 1101 | 5 |
| 1110 | 6 |
| 1111 | 7 |

Note: in our representation the bias is 0, so $0 = 10\ldots0$. In some versions the bias is 1, so $1 = 10\ldots0$.

If we write the bit strings in alphanumeric order as above then 0 will be the bit string just below the middle, with the positive numbers below in ascending order and the negative above in decreasing order.

Of course, we can use the excess representation for any number, $K$, of bits. For a binary sequence of $K$ bits, first remove the leading (left-most) bit. Transform the remaining bits into the correspondong decimal number, $n$ say. If the original leading bit was 1 then the final decimal number is $n$. If the original leading bit was 0 then the final decimal number is $n - 2^{K-1}$. So

$$1b_{K-2}\ldots b_1b_0 = (b_{K-2}\ldots b_1b_0)_2 \qquad 0b_{K-2}\ldots b_1b_0 = (b_{K-2}\ldots b_1b_0)_2 - 2^{K-1}$$

For 101001 we have 01001 so $n = 2^3 + 2^0 = 9$, and the final decimal is 9.

For 011111 we have 11111 so $n = 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 31$ and the final decimal is $n - 2^5 = 31 - 32 = -1$. Notice, $-1 \equiv_{32} 31$.

For 001101 we have $n = 01101_2 = 13$ so the final decimal number represented is $13 - 32 = -19$.

In reality the representation is a form of modular arithmetic in which $b_Kb_{K-1}\ldots b_1b_0$ represents $(b_Kb_{K-1}\ldots b_1b_0)_2 - 2^{K-1}$.

### 2.3.5 Two's complement notation

Although the excess representation avoids the wasted double representation of 0, it still has the issue that the representation is somewhat unnatural and arithmetic is not simple modular arithmetic. An alternative is to take the modular arithmetic representation to its natural conclusion and use the negative number equivalents of half the numbers.

We describe this first for 4-bit numbers. In the natural power-of-2 based representation, $1101_2 = 2^3 + 2^2 + 2^0 = 13$. In two's complement it represents $-3$, which is the same as 13 modulo $2^4 = 16$. In general the first half of the binary numbers represent themselves and the second half represent an equivalent negative number modulo 16.

| Bit pattern | Signed integer value (two's complement notation) |
|---|---|
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 2 |
| 0011 | 3 |
| 0100 | 4 |
| 0101 | 5 |
| 0110 | 6 |
| 0111 | 7 |
| 1000 | −8 |
| 1001 | −7 |
| 1010 | −6 |
| 1011 | −5 |
| 1100 | −4 |
| 1101 | −3 |
| 1110 | −2 |
| 1111 | −1 |

For bits strings of length $K$, the two's complement representation is defined as follows:
If the left-most bit is 0 then the number is the usual binary representation. This gives the numbers from 0 to $2^{K-1} - 1$.
If the left-most bit is 1 then the number is $(\textit{usual binary representation}) - 2^K$. This gives the numbers from $-2^{K-1}$ to $-1$.

Formally, for a bit string $b_{K-1} \dots b_1 b_0$, the number represented is

$$b_{K-1}2^{K-1} + \ldots + b_1 2^1 + b_0 2^0, \quad \text{if } b_{K-1} = 0 \quad \text{and}$$

$$(b_{K-1}2^{K-1} + \ldots + b_1 2^1 + b_0 2^0) - 2^K, \quad \text{if } b_{K-1} = 1$$

Equivalently

$$\begin{array}{lcccl} 0b_{K-2} \dots b_1 b_0 & = & (0b_{K-2} \dots b_1 b_0)_2 & = & (b_{K-2} \dots b_1 b_0)_2 \\ 1b_{K-2} \dots b_1 b_0 & = & (1b_{K-2} \dots b_1 b_0)_2 - 2^K & = & (b_{K-2} \dots b_1 b_0)_2 - 2^{K-1} \end{array}$$

Notice, you can tell the sign of a two's complement number from its leading, left-most, bit. But the sign is + if this bit is 0 and − if the bit is 1, the opposite of the case for the excess representation.

Since $2^K = 0$ modulo $2^K$, the negation of $x$ is $y$ where $x+y = 2^K$. In two's complement, if $y \geq 2^{K-1}$ then we take it to be $-x$ instead. For example, $21 + 43 = 2^6$ so in a 6-bit representation 101011 represents -21 rather than 43.

So to find the decimal version of a number in two's complement representation when the left-most digit is 1, start at the left and invert all the digits before the last (right-most) 1 and then take the negation of the decimal version of the result.

## Finding negations

There are three simple ways of finding the negation of a number represented using two's complement. For the last two, negations are found by inverting (complementing mod 2) the bits.

- ◇ for a $K$-bit number, subtract the number from $2^K$
- ◇ invert all the bits from the left up to, but excluding, the right-most 1
- ◇ invert all the bits and then add $00\ldots001$ to the result

The first method works by modular arithmetic.

To see that the second method works note that, for any binary number of length $K$, swapping all the bits up to but excluding the right-most 1, and adding the two numbers gives $2^K$.

$1111 + 0001 = 10000$

$01010101 + 10101011 = 100000000$

$0101010 + 1010110 = 10000000$

One significant advantage of two's complement is that subtraction can easily be implemented using bit inversion and addition. With the second method of negation we get

$3 - 5$ becomes $0011 - 0101$ which is
$0011 + (-0101) = 0011 + 1011 = 1110 = -(0010) = -2$

For the third method of negation above, consider a bit string $b_K \dots b_{i+1} b_i \dots b_1$ where $b_i$ is the left-most 1. We have

$$b_K \dots b_{i+1} 011 \dots 1 \; + \; 0 \dots 01 \quad = \quad b_K \dots b_{i+1} 100 \dots 0$$

Inverting all the bits gives $\overline{b_K} \dots \overline{b_{i+1}} 01 \dots 1$. Adding $00 \dots 01$ gives $\overline{b_K} \dots \overline{b_{i+1}} 10 \dots 0$ which is the string constructed according to method 2. So we can also find negations of numbers by inverting all the bits then adding $0 \dots 01$, allowing subtraction to be implemented using full inversion and then two additions, rather than testing for the rightmost 1.

$3 - 5$ is $0011 + (-0101) = 0011 + 1010 + 0001 = 1110 = -(0010) = -2$

## Two's complement arithmetic

We define addition and multiplication in two's complement numbers as though they were unsigned numbers, i.e. positionly.

$$3 + 2 = 0011 + 0010 = 0101 = 5$$

In fact this gives us $+$, $-$ and $*$ modulo $2^n$.

$$3 + 8 = 0011 + 1000 = 1011 = -5 \; (= \; 11 \; mod \; 16)$$

Modular arithmetic was introduced in CS1860. Two integers $a$ and $b$ are *equivalent* mod $2^n$ if $2^n$ divides $a - b$. Non-negative numbers are equivalent mod $2^n$ if their binary representations, padded with leading 0s, have the same last $n$ bits.

We define $+$, $-$ and $*$ as for unsigned numbers.
$4 + 3 = 0100 + 0011 = 0111 = 7 \qquad 2 + (-5) = 0010 + 1011 = 1101 = -3$
$5 - 2 = 0101 - 0010 = 0011 = 3 \qquad 7 + 3 = 0111 + 0011 = 1010 = -6 (= 10 mod 16)$
$-3 \times -2 = 1101 \times 1110 = 11010 + 110100 + 1101000 = 1110110 = 0110 = 6$
This is arithmetic modulo $2^n$.

Here is a comparison of the three representations for 4-bit numbers.

| Bits | Sign-magnitude | Excess | Two's complement | bit inversion |
|---|---|---|---|---|
| 0000 | 0 | $-8$ | 0 | |
| 0001 | 1 | $-7$ | 1 | |
| 0010 | 2 | $-6$ | 2 | |
| 0011 | 3 | $-5$ | 3 | |
| 0100 | 4 | $-4$ | 4 | |
| 0101 | 5 | $-3$ | 5 | |
| 0110 | 6 | $-2$ | 6 | |
| 0111 | 7 | $-1$ | 7 | |
| 1000 | 0 | 0 | $-8$ | 1000 |
| 1001 | $-1$ | 1 | $-7$ | 0111 |
| 1010 | $-2$ | 2 | $-6$ | 0110 |
| 1011 | $-3$ | 3 | $-5$ | 0101 |
| 1100 | $-4$ | 4 | $-4$ | 0100 |
| 1101 | $-5$ | 5 | $-3$ | 0011 |
| 1110 | $-6$ | 6 | $-2$ | 0010 |
| 1111 | $-7$ | 7 | $-1$ | 0001 |

**Exercise** Suppose $n = 6$. What is the two's complement representation of (i) 27 and (ii) $-23$?

What number is represented by 110100 in the two's complement notation?

## 2.4 Ranges and overflow

Suppose we have $n$ bits to store an integer.

For two's complement notation

- ◇ The largest positive value that can be stored is $011\ldots11_2$ (with $n-1$ ones), which is $(10\ldots00_2) - 1$ (with $n-1$ zeros), which is $2^{n-1} - 1$.
- ◇ The largest absolute value of a negative number that can be stored is $10\ldots00_2$ (with $n-1$ zeros), which is $2^{n-1}$.

So the range is from $-2^{n-1}$ to $2^{n-1} - 1$.

For unsigned $n$-bit integers the range is 0 to $2^n - 1$.

For excess notation $n$-bit integers the range is $-2^{n-1}$ to $2^{n-1} - 1$.

For sign magnitutde notation $n$-bit integers the range is $-2^{n-1} + 1$ to $2^{n-1} - 1$.

The following table shows the ranges for four integer data types in Java.

| Name | Size | Range |
|---|---|---|
| byte | 1 | $-128$ ... 127 |
| short | 2 | $-32,768$ ... $32,767$ |
| int | 4 | $-2,147,483,648$ ... $2,147,483,647$ |
| long | 8 | $-9,223,372,036,854,775,808$ ... $9,223,372,036,854,775,807$ |

The second column is the size in bytes, the third column are the top and bottoms of the ranges, written as decimal numbers, $128 = 2^8$; $32,768 = 2^{16}$; $2,147,483,648 = 2^{32}$; $9,223,372,036,854,775,808 = 2^{64}$.

We say that an integer is *in range* if it is in the range determined by the given representation. Every integer in the range is the value of some bit pattern.

### 2.4.1 Overflow

*Overflow* occurs if the result of an arithmetic operation is outside the range of numbers which can be represented in the chosen representation. For example, in an unsigned 5-bit representation the largest number available is $2^5 - 1 = 31$ so adding 23 and 12 will cause an overflow. For a 5-bit signed representation the range is $-16$ to 15 so $8 + 10$ and $(-4) - 13$ will cause overflow but $4 - 13$ will not.

In general, for an $n$-bit representation, if the result of an operation on two numbers is outside the range which can be represented the right-most $n$ bits will be kept and the remaining left-most bits will be discarded. As a result the calculation will give a result in the range, but it will not be what was expected.

**Example** In a 4-bit unsigned representation, adding 0101 (i.e. 5) and 1100 (i.e. 12) gives 10001 which is stored as 0001, i.e. 1 rather than 17.

For two's complement the range of numbers which can be represented is $-2^{n-1}, \ldots, 2^{n-1} - 1$. For $n = 4$, in two's complement 1100 represents $-4$ so $0101 + 1100$ represents $5 + (-4)$ and the answer $0001 = 1$ is correct, and within the range $-8$ to 7. Thus overflow has not occurred.

However, in two's complement, $0100 + 0101$ represents $4 + 5 = 9$ which is out of range. The result is 1001, which represents $-7$ (this is 9 mod 16), and $4 + 5 = -7$ is not usually what the programmer expects.

Even negating a two's complement number can lead to an overflow if the number is at the bottom of the range. If we negate 1000 (i.e. $-8$) we again obtain 1000 (i.e. $-8$).

Overflow and its consequences are thus an important issue, requiring knowledge of the number representation being used, and programmers must write their programs accordingly. This is sometimes referred to as 'dealing with potential overflow'.

The importance of the need to understand and allow for overflow is illustrated by the crash of Arianne 5 Flight 501. The programmers neglected to correctly allow for overflow when some code was converted from 64-bit floating point format to 16-bit signed integer format. This is one of the most expensive software bugs ever.

## 2.5 Representing real numbers

We can express many fractional numbers using a decimal point. In the standard base 10 number system digits to the right of the decimal point are coefficients of negative powers of 10.

$$37.687 \;=\; (3 \times 10) + (7 \times 10^0) + (6 \times 10^{-1}) + (8 \times 10^{-2}) + (7 \times 10^{-3})$$

We can do the same thing with binary numbers

$$01101.011 \;=\; (0\times 2^4)+(1\times 2^3)+(1\times 2^2)+(0\times 2^1)+(1\times 2^0)+(0\times 2^{-1})+(1\times 2^{-2})+(1\times 2^{-3})$$

$$= \; 8 + 4 + 1 + \frac{1}{4} + \frac{1}{8} \;=\; 13.375$$

The issue is that with a binary string representation, how do we know where the 'decimal point' is?

One possibility is to simply assert that, say, the point is before the last four digits. So 11011010=1101.1010. This is called fixed point notation.

However, this gives the same degree of 'precision' to all numbers. If we are measuring, say, distances then a few 100 miles here or there when talking about the distance away of objects in our solar system is less important than even half or quarter of a mile if we are carrying a heavy load from the shops. In other words, we may want the level of precision to reflect the size of the number.

We can write fractional numbers in a standard form with, say, 0 before the decimal point and a multiplication by a power of 10.

$$37.687 \;=\; 10^2 \times 0.37687$$

We can do the same thing with base 2.

$$110.1011 \;=\; 2^3 \times 0.1101011$$

However, we still need to specify the power of 2 and the sign of the number. So, instead of all the bits being part of the actual number, one bit is used for the sign, some of the bits are used to define the power of 2, and the remaining bits are taken to represent a number with the decimal point at the extreme left.

This is a *floating-point* representation. It has a sign, e.g. +, a *base*, e.g. base 2, an *exponent*, the power to which that base is raised e.g 4, and a nonnegative *mantissa*, or *significand*, the actual number e.g. 0.1011.

$$number = \pm\, base^{exponent} \times mantissa$$

For a binary bit string representation, the first bit corresponds to the sign, 0 for + and 1 for −, then next $k$ digits are the exponent and the rest of the digits are the mantissa.

For 8-bit representations it is usual to take $k = 3$ and $n = 4$, for 32-bits we have single precision in which $k = 8$ and $n = 23$ and for 64-bits we have double precision in which $k = 11$ and $n = 52$.

The mantissa is always non-negative but the exponent can be either positive or negative. We use an unsigned representation for the mantissa, and we shall use the excess representation for the exponent.

For example, under the floating point representation

$$01101011 = +2^{110} \times (2^{-1} + 2^{-3} + 2^{-4}) = +2^2 \times \frac{11}{16} = 2.75$$

Notice, this gives a different number to the fixed point interpretation.

**Examples**

00100110: The sign bit is 0, the exponent bits are 010, and the mantissa bits are 0110. The exponent is given in the excess notation, which means that the actual exponent is $010_2 - 100_2 = 2 - 4 = -2$. The mantissa is 0110 with the decimal point at the extreme left, `.0110` (or `0.0110` if you find that easier to read) which is the decimal $1/4+1/8 = 3/8$. The sign bit 0 is interpreted as +. Therefore, the number represented is $+2^{-2} \times (3/8) = 3/32$.

11101100: The sign bit is 1, the exponent bits are 110, and the mantissa bits are 1100. The exponent is $110_2 - 100_2 = 6 - 4 = 2$. The mantissa is `.1100`, i.e. the decimal $1/2 + 1/4 = 3/4$. The sign bit 1 means −. Therefore, the number represented is $-2^2 \times (3/4) = -3$.

Represent +7/2: The sign bit is 0. Then

$$\frac{7}{2} = 3 + \frac{1}{2} = 2^1 + 2^0 + 2^{-1} = 2^2 \times (2^{-1} + 2^{-2} + 2^{-3})$$

so the exponent is 2, which is 110 in excess notation, and the mantissa is 1110 so we get 01101110.

Represent $-5/16$: The sign bit is 1. Then

$$\frac{5}{16} = \frac{1}{4} + \frac{1}{16} = 2^{-2} + 2^{-4} = 2^{-1} \times (2^{-1} + 2^{-3})$$

so the exponent is $-1$, which is 011 in excess notation, and the mantissa is 1010 so we get 10111010.

**Exercise** What real numbers are represented by the following bit patterns under the floating point representation: (i) 01000111 (ii) 10011110

## 2.6 Further reading

- ◇ Brookshear, Chapter 1 (except for the first 2 or 3 sections, depending on the edition)
- ◇ A lot of interesting (and advanced) information in
  Donald Knuth, *The Art of Computer Programming*, volume 2 "Seminumerical Algorithms", Chapter 4 "Arithmetic". Third edition. Addison-Wesley, 1997

# Chapter 3

# Propositional Logic

Logic has many applications in computer science. For example, in formal reasoning, the design and verification of algorithms (which form the basis of all programs), in the design of logic and switching circuits, in the specification of processes, and in a programming methodology called logic programming, PROLOG. Of course, logical expressions appear in programming language constructs after, for examples, `if` and `while` statements. In this course we introduce the basic concepts of propositional and predicate logic, and discuss the meaning and methods of proof.

Most of the material in this chapter can be found in Sections 1.1, 1.2, 1.3, and 3.1 of Rosen which you are strongly urged to read.

Logic consists of a set of rules for drawing inferences. We assume that certain statements, called *axioms*, are true and we have a set of rules for proving consequences of the axioms.

For example, we could have axioms

Henry VIII was the father of Elizabeth I
Mary Rose was the sister of Henry VIII

and the rule

If (A was the father of B) and (C was the sister of A) then (C was the aunt of B).

Using these rules and axioms we can deduce that Mary Rose was the aunt of Elizabeth I.

In integer arithmetic we have axioms and rules

$$x + y = y + x$$
$$x + 0 = x$$
$$\text{if } A = B \text{ then } B = A$$
$$\text{if } (A = B \text{ and } B = C) \text{ then } A = C.$$

Then we can deduce that $x = 0 + x$.

## 3.1 Propositions

A *proposition* is a statement which is either true or false. For example,

CS1870 is a first year course at Royal Holloway.
A comes before B in the English alphabet.
$99 > 6$

are all statements. A statement which is true is said to have *truth value* true, and a false statement is said to have *truth value* false. The above statements all have truth value true.

CS1870 is a second year course at Royal Holloway.
A does not come before B in the English alphabet.
$6 = 2 \times 4$

are all statements whose truth value is false.

There are propositions which are either true or false, but we don't know which. For example,

There are life forms outside of earth's solar system.
On 1st November 2062 there will be an earthquake in India.
Given any integer $N$, there are primes $p$ and $p + 2$ greater than $N$.

Logic is concerned with propositions but not with absolute truth values. *Axioms* are propositions which are *assumed* to be true. We use logic to determine consequences of axioms. We do not *prove* that the axioms are true, but if the axioms are true then so are the consequences. For example:

if I miss the train, then I shall be late for work.

I will be late for work will be true if I miss the train, but if I don't miss the train then I may or may not be late for work, we can't tell.

if it is true that for any $N$, there are primes $p$ and $p + 2$ greater than $N$,
then there are infinitely many pairs of primes of the form $(p, p + 2)$.

**Goldbach's conjecture**: every even number greater than 2 is the sum of two primes.

If Goldbach's conjecture is true, we can write $2n$ in the form $p + q$, where $p, q$ are primes.

**Exercise** Which of these are propositions?
1. I am happy.
2. What is your name?
3. It is false that grass is red.
4. 8 is a prime number.
5. Close the door!
6. If I have a cold, then I sneeze.
7. If 2+2=5 then grass is usually red.
8. Every even number greater than 2 is the sum of two primes.

Some sentences are not propositions because they cannot have a truth value. For example,
9. This sentence is false.

A sentence which cannot be a proposition is called a *paradox*. We often use $P, Q, R$ etc. for propositional variables which can have truth value T (true) or F (false).

## 3.2 Logical Operations

We can join propositions together to get new propositions. We can construct compound propositions by applying logical operations (*connectives*). For example,

Henry V was King of England and 5>3

is a proposition which is true.

We often use letters such as $P, Q, R$ for propositional variables which can have truth value $T$ (true) or $F$ (false).

There is a set of logical operations which can be used to construct new propositions from existing ones, and there is a set of rules which determine the truth values of the new propositions. These connectives are called **and** , **or** , and **implies** . There is also an operator called **not**. The standard mathematical notation for these operations is $\wedge$, $\vee$, $\Rightarrow$, and $\neg$ respectively, and the study of these is known as propositional calculus.

We define each operation by stating the truth value of the new proposition in terms of its components. For convenience these values are given as a *truth table*. The truth table lists a value of a compound proposition for all possible values of it components.

**NOT**, $\neg$ (negation)

For any proposition P, the truth value of $\neg P$ is the opposite of the truth value of P. Thus the truth table for $\neg$ is

| P | $\neg$ P |
|---|---|
| T | F |
| F | T |

So for example

$\neg$(14>6) is false
$\neg$(Obama is Prime Minister of the UK) is true
$\neg$(Obama has been President of the US) is false

**AND**, $\wedge$ (conjunction)

P **and** Q is true if both P and Q are true, and false otherwise.

| P | Q | P $\wedge$ Q |
|---|---|---|
| T | T | T |
| T | F | F |
| F | T | F |
| F | F | F |

So for example

2 is an even prime number, i.e. (2 is even)$\wedge$(2 is prime) is true
(Jupiter is a planet) **and** (a week has 7 days) is true
(9.3 is positive)$\wedge$(9.3 is an integer) is false

**OR**, $\vee$ (disjunction)

P **or** Q is true if either P or Q is true, and false otherwise.

| P | Q | P ∨ Q |
|---|---|---|
| T | T | T |
| T | F | T |
| F | T | T |
| F | F | F |

So for example

(14>6)∨(4 exactly divides 8) is true
(The moon is made of green cheese)∨(Grass is red) is false

In mathematics, 'or' always means inclusive: at least one of, but maybe both. There is a special operator for exclusive or called **xor**. We'll meet this later.

**IMPLIES**, ⇒ (implication)

This connective is equivalent to the programming language statement 'if P then Q'. P⇒Q says that if P is true then Q must be true. So if P is true and Q is false then P⇒Q is false. The proposition does not say anything about the case when P is false, if P is false then there is no requirement on Q. So P⇒Q is always true when P is false.

| P | Q | P ⇒Q |
|---|---|---|
| T | T | T |
| T | F | F |
| F | T | T |
| F | F | T |

So for example

(9>5) ⇒(240 is an even number)
(4 exactly divides 5) ⇒(14>6) is true
(4 exactly divides 5) ⇒ **not**(14>6) is true
(The moon is made of green cheese) ⇒ (Grass is red) is true.

Note, this is not the same as cause and effect.

**LOGICALLY EQUIVALENT**, ⇔ (equal)

This says that $P$ and $Q$ have the same truth value.

| P | Q | P ⇔ Q |
|---|---|---|
| T | T | T |
| T | F | F |
| F | T | F |
| F | F | T |

($N$ is an even number) ⇔ ($N$ is divisible by 2) is true
($N$ is an odd number) ⇔ ($N$ is divisible by 2) is false

**Exercises** Given propositions

$P$ = logic is fun
$Q$ = today is Friday
$R$ = the sun is shining

Express the following as compound propositions:
1. Logic is not fun and today is Friday.
2. Today isn't Friday, nor is logic fun.
3. Either logic is fun or it's Friday and the sun is shining.
4. If the sun is shining and it's not Friday then logic isn't fun.

Of course, we can build up propositions using several operators. For example,

$$\mathrm{P} \wedge (\mathrm{Q} \Rightarrow \mathrm{R}), \quad \neg(\mathrm{P} \vee (\mathrm{Q} \wedge \mathrm{P}))$$

These operations have precedence, $\neg$ is highest, $\wedge$ and $\vee$ are equal precedence, and $\Rightarrow$and $\Leftrightarrow$ are of equal and lowest precedence.

We can use truth tables to give the truth values of a complex proposition in terms of its components.

| P | Q | R | Q⇒R | P∧(Q⇒R) |
|---|---|---|---|---|
| T | T | T | T | T |
| T | T | F | F | F |
| T | F | T | T | T |
| T | F | F | T | T |
| F | T | T | T | F |
| F | T | F | F | F |
| F | F | T | T | F |
| F | F | F | T | F |

**Exercises** Draw the truth tables for (i)$\neg P \vee Q$, (ii) $(R \wedge \neg Q) \Rightarrow \neg P$, (iii) $P \Rightarrow (Q \wedge \neg P)$, (iv) $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$

In any formal arena, such as engineering, design, computer programming or mathematics, it is important to give a precise definition of the terms being used. We now give a formal definition of what it means for two propositions to be equal.

**Definition** Two propositions are *equal* if they always have the same truth values. We can use truth tables to prove that propositions are (or are not) equal.

If $P$ and $Q$ have the same truth values, we often say that they are *logically equivalent* and write $P \Leftrightarrow Q$.

For example, it is easy to check that both $\neg$(P$\wedge$Q) and ($\neg$P)$\vee$($\neg$Q) have truth table

| P | Q | |
|---|---|---|
| T | T | F |
| T | F | T |
| F | T | T |
| F | F | T |

Thus we have that $\neg(P \wedge Q) \Leftrightarrow (\neg P) \vee (\neg Q)$.

Some propositions are always true. For example,

| P | Q | P∨Q | P ⇒(P∨Q) |
|---|---|---|---|
| T | T | T | T |
| T | F | T | T |
| F | T | T | T |
| F | F | F | T |

so we see that $P \Rightarrow (P \vee Q)$ is always true, as is $P \vee \neg P$.

| P | $\neg$P | P $\vee \neg$P |
|---|---|---|
| T | F | T |
| F | T | T |

Propositions that are always true are called *tautologies*.

It is also the case that some propositions are always false. For example,

| P | $\neg$P | P$\wedge \neg$P |
|---|---|---|
| T | F | F |
| F | T | F |

Propositions that are always false are called *contradictions*. Propostions which are neither tautologies or contadictions are *contingent*.

We complete this introduction to propositional logic by returning to the 'exclusive or' operation mentioned above. We expect P **xor** Q to be true if exactly one of P or Q is true. So we expect a truth table of the form

| P | Q | P **xor** Q |
|---|---|---|
| T | T | F |
| T | F | T |
| F | T | T |
| F | F | F |

Considering the truth table for $(P \wedge \neg Q) \vee (\neg P \wedge Q)$, we see that, as we might expect

$$(P \wedge \neg Q) \vee (\neg P \wedge Q) = P \textbf{ xor } Q \ .$$

| P | Q | $(P \wedge \neg Q)$ | $(\neg P \wedge Q)$ | $(P \wedge \neg Q) \vee (\neg P \wedge Q)$ |
|---|---|---|---|---|
| T | T | F | F | F |
| T | F | T | F | T |
| F | T | F | T | T |
| F | F | F | F | F |

## 3.3 Normal forms

We can write propositional expressions in uniform ways.

### 3.3.1 Disjunctive normal form

A formula is said to be in *disjunctive normal form* (DNF) when it is a disjunction ($\vee$) of conjunctions ($\wedge$) of propositional variables or their negations. For example:

$$(P \wedge \neg Q \wedge R) \vee (\neg Q \wedge \neg R) \vee Q$$

Every expression built up according to the rules of propositional calculus is equivalent to some formula in disjunctive normal form.

We can construct a DNF for a proposition from its truth table.

1. For each row whose truth value is *true*, write down, for each of the propositional variables $P_i$ in the formula, either $P_i$ if true in row or $\neg P_i$ if false. Then take the conjunction of these expressions.

2. Repeat 1 for each row in the truth table where the value is true and write down the disjunction of all the conjunctions.

The result is a formula in DNF which is equivalent to the original formula.
(NB A DNF is not necessarily unique)

**Example** P $\wedge$ (Q$\Rightarrow$R)

The truth table for this expression is given above.
For the first row, $P = Q = R = T$ so we have $P \wedge Q \wedge R$.
The second row is false. For the third row we have $P = R = T$, $Q = F$ so we have $P \wedge \neg Q \wedge R$.
For the fourth row we have $R = Q = F$, $P = T$ so we have $P \wedge \neg Q \wedge \neg R$.
The other rows are false, so we have

$$(P \wedge Q \wedge R) \vee (P \wedge \neg Q \wedge R) \vee (P \wedge \neg Q \wedge \neg R)$$

**Exercises** Find a DNF for (i) $P \Rightarrow (Q \wedge \neg R)$ (ii) $P \Rightarrow (Q \wedge \neg P)$

### 3.3.2 Conjunctive normal form

A formula is said to be in *conjunctive normal form* (CNF) when it is a conjunction ($\wedge$) of disjunctions ($\vee$) of propositional variables or their negations. For example:

$$(\neg P \vee Q \vee R \vee \neg S) \wedge (P \vee Q) \wedge \neg S \wedge (Q \vee \neg R \vee S)$$

Every expression built up according to the rules of propositional calculus is equivalent to some formula in conjunctive normal form. To construct a CNF of an expression we use logical equivalences.

### 3.3.3 Some logical equivalences

Suppose that $P$, $Q$ and $R$ are propositional expressions. The following logical equivalences can all be proved using truth tables.

**Double negation**
$\neg(\neg P) \Leftrightarrow P$

**Commutative Laws**
$P \vee Q \Leftrightarrow Q \vee P$
$P \wedge Q \Leftrightarrow Q \wedge P$

**Associative Laws**
$(P \vee Q) \vee R \Leftrightarrow P \vee (Q \vee R)$
$(P \wedge Q) \wedge R \Leftrightarrow P \wedge (Q \wedge R)$

**Distributive Laws**
$P \vee (Q \wedge R) \Leftrightarrow (P \vee Q) \wedge (P \vee R)$
$P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$

**De Morgan's Laws**
$\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$
$\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$

**Implication**
$(P \Rightarrow Q) \Leftrightarrow (\neg P \vee Q)$
$(P \Leftrightarrow Q) \Leftrightarrow (P \Rightarrow Q) \wedge (Q \Rightarrow P)$

**Excluded middle**
$P \vee \neg P \Leftrightarrow T$

**Contradiction**
$P \wedge \neg P \Leftrightarrow F$

**Idempotence**
$P \vee P \Leftrightarrow P$
$P \wedge P \Leftrightarrow P$

**Identity**
$P \wedge T \Leftrightarrow P$
$P \vee F \Leftrightarrow P$

**Domination**
$P \vee T \Leftrightarrow T$
$P \wedge F \Leftrightarrow F$

### 3.3.4 Using logical equivalences to obtain a normal form

The general approach is to rewrite an expression to one that is logically equivalent until an expression in CNF, or DNF, is constructed.

The following strategy is likely to be helpful when trying to decide which equivalences to use.

1. Use Implication Laws to eliminate $\Rightarrow$ and $\Leftrightarrow$
2. Use Double Negation and De Morgan to bring $\neg$ immediately before propositional variables
3. Repeatedly use distributive laws (and, optionally, other laws) to obtain a normal form.

**Example** Using logical equivalences, find a DNF and CNF for $Q \Rightarrow (\neg P \wedge (Q \vee R))$.

$$\begin{aligned} Q \Rightarrow (\neg P \wedge (Q \vee R)) &\Leftrightarrow \neg Q \vee (\neg P \wedge (Q \vee R)) \\ &\Leftrightarrow \neg Q \vee ((\neg P \wedge Q) \vee (\neg P \wedge R)) \\ &\Leftrightarrow \neg Q \vee (\neg P \wedge Q) \vee (\neg P \wedge R) \end{aligned}$$

This gives a DNF. We can then apply more equivalences

$$\begin{aligned} Q \Rightarrow (\neg P \wedge (Q \vee R)) &\Leftrightarrow \neg Q \vee (\neg P \wedge (Q \vee R)) \\ &\Leftrightarrow (\neg Q \vee \neg P) \wedge (\neg Q \vee (Q \vee R)) \\ &\Leftrightarrow (\neg Q \vee \neg P) \wedge (\neg Q \vee Q \vee R) \end{aligned}$$

to get a CNF. Of course, in this case there is a simpler CNF, $\neg Q \vee \neg P$ which is also a DNF.

**Exercise** Find a DNF and CNF for $P \Rightarrow (Q \wedge \neg R)$.

If you are not sure whether your answer is correct you can always check by writing out the truth tables.

Don't forget, any disjunction ($\vee$) of propositional variables or their negations is both a DNF and a CNF, and any conjunction ($\wedge$) of propositional variables or their negations is both a DNF and a CNF.

**Exercise** Using logical equivalences, show that the following expression is a tautology.
$(\neg Q \wedge (P \Rightarrow Q)) \Rightarrow \neg P$

Sometimes these more general distributive laws are useful:

$$\begin{array}{lcl} (A \wedge B) \vee (C \wedge D) & \Leftrightarrow & (A \vee C) \wedge (A \vee D) \wedge (B \vee C) \wedge (B \vee D) \\ (A \vee B) \wedge (C \vee D) & \Leftrightarrow & (A \wedge C) \vee (A \wedge D) \vee (B \wedge C) \vee (B \wedge D) \end{array}$$

They follow from the usual distributive laws.

**Exercise** Using logical equivalences, find DNF and CNF of $\neg(P \Rightarrow Q) \vee (\neg P \wedge \neg Q)$.

## 3.4 Practice exercises

1. Assume that P and Q are true propositions, and that R and S are false propositions. Determine the truth values of
   a. P∧Q
   b. P∨S
   c. P⇒Q
   d. R ⇒ ¬S.

2. Construct the truth table for P ∨ ¬Q, and use it to determine the value of P ∨ ¬Q when P and Q are both false.

3. Let P be the proposition 'my umbrella is at home' and let Q be the proposition 'it will rain today'. Describe each of the following propositions in words:
   a. P **and** Q
   b. P **or** Q
   c. P ⇒ Q.

4. Identify the propositions in the following sentences, and rewrite the sentences using mathematical notation.
   a. Mary is John's mother and Michael is six.
   b. Mary is not John's mother or Michael is six.

5. Let P, Q, R be the following propositions
   P: John is 5
   Q: Michael is 6
   R: Mary is John's mother
Write each of the following sentences in mathematical notation, using P,Q,R and the logical operators.
   a. John is 5, and Mary is not John's mother.
   b. If Michael is 6, then John is 5.
   c. If John is not 5, then it is false that Michael is 6.
   d. It is false that Mary is John's mother, and that Michael is 6.
   e. Mary is not John's mother or Michael is not 6.

6. Construct truth tables for the following propositions.
   a. ¬(P∧Q)
   b. ¬(P∨(Q∧P))
   c. (P ∧ ¬Q) ⇒ F
   d. (P∧Q)∧R
   e. P∧(Q∧R)
   f. R ⇒ (¬P ⇒ Q)

7. Use truth tables to show that (((P∧Q) ∨ ¬P) ∨ ¬Q) is a tautology.

8. Use logical equivalences to show that (((P∧Q) ∨ ¬P) ∨ ¬Q) is a tautology.

9. Find a DNF and a CNF for $(((P \wedge (Q \Rightarrow R)) \wedge \neg R) \vee \neg(Q \wedge \neg P))$.

# Chapter 4

# Predicate Logic and Proof

The material on predicates discussed in this chapter can be found in Section 1.3 of Rosen.

So far we have only considered simple propositions involving specific objects. Propositional logic is not sufficient to express all sentences or arguments. For example, "Peter likes football" is a proposition which may be true or false, and we can write it as likes(peter,football). (This is a relation as defined in CS1860.) But how do we write sentences such as

- Everyone likes football
- Some people like cats
- Someone has stolen my car
- Every student is doing some course

And how do we work out whether such sentences are true or false?

## 4.1 Predicates

To deal with real applications we need to have propositions which make statements about collections of objects. For example, 'all integers which are divisible by 4 are even' and 'there exists an integer $n$ such that $n^2 = 36$'.

- variables to stand for individual objects
- quantifiers to express how many objects are involved
- domain of discourse to define the range of objects

A statement about a property of an object is called a *predicate*. We use the notation $P(x)$ to denote a statement which concerns properties of the object $x$.

The set of values over which the quantifier ranges is called the *domain of discourse*.

For "Everyone likes football" and for "Some people like cats" the domain of discourse is the set of all people. The first sentence means all members of the set like football, the second is saying at least one of the domain like cats, but not necessarily everyone.

In the following examples the domain of discourse is the set of integers. For example, we may have

$$P(n) = ((n \text{ divisible by } 4) \Rightarrow (n \text{ even})),$$

where $n$ is an integer.

Then we write the statement 'all integers which are divisible by 4 are even' as 'for all integers $n$, $P(n)$'.

Another example: $P(n) = (n^2 = 36)$.

Then we write the statement 'there is an integer such that $n^2 = 36$' as 'there exists an integer $n$ such that $P(n)$'.

## 4.2 Universal and existential quantifiers

There are two cases: $P(x)$ may be true for all choices of $x$ from a particular set, or $P(x)$ may be true for some choice of $x$ from the set. (The case when $P(x)$ is never true is not interesting.) There are two symbols, called *predicate quantifiers*, which are used to denote these two cases.

**Universal quantifier**

The symbol $\forall$, called *for all*, is used to indicate that a statement applies to all choices of object.

$\forall x\ P(x)$ – for all $x$, $P(x)$ is true.

If the domain of discourse is the set of all people, "Everyone likes football" can be written $\forall x\ likes(x, football)$ or as $\forall x\ (person(x) \Rightarrow likes(x, football))$.

**More Examples**

Domain of discourse: set of all people, "Everyone on the escalator must wear shoes" could be written $\forall x\ (on\_escalator(x) \Rightarrow wears(x, shoes))$.

Domain of discourse: $\mathbb{Z}$, "All integers which are divisible by 4" could be written $\forall n\ (divisible(n, 4) \Rightarrow even(n))$.

Domain of discourse: set of all cats, "All cats are friendly and have four legs" could be written $\forall x\ (friendly(x) \Rightarrow has\_legs(x, 4))$.

**Exercises** For the following suggest a suitable domain of discourse and then express each as a formula using the universal quantifier:
(i) All horses eat hay
(ii) Dogs have teeth
(iii) The square of an odd number is odd.

**Existential quantifier**

The symbol $\exists$, called *there exists*, is used to indicate that a statement applies to at least one choice of object.

$\exists x\ P(x)$ – there exists an $x$ such that $P(x)$ is true.

When

$$P(n) = ((n \text{ divisible by } 4) \Rightarrow (n \text{ even}))$$

then $\forall n\ P(n)$, is the statement 'all integers which are divisible by 4 are even'. Strictly speaking we should include the set that $n$ ranges over, so we would write

$$\forall(\text{integers } n)P(n),$$

but this becomes complicated so it is more usual to state the domain of discourse,

$$\forall n P(n), \quad \text{where } n \text{ is an integer.}$$

**More Examples**

Domain of discourse: set of all people, "Some people like cats" could be written $\exists x\ likes(x, cats)$.

Domain of discourse: set of all people, "Someone has stolen my car" could be written $\exists x$ $stolen(x, my\_car)$.

Domain of discourse: $\mathbb{Z}$, "Some integer is equal to $4^2$" could be written $\exists n$ $(n = 4^2)$.

Like all propositions, a predicate may be true or false.
$\exists x P(x)$ is **true** if there is at least one $x$ in the domain of discourse with $P(x)$ is true.
$\exists x P(x)$ is **false** if $P(x)$ is false for all possible values of $x$.
$\forall x P(x)$ is **true** if $P(x)$ is true for all possible values of $x$.
$\forall x P(x)$ is **false** if there is at least one value of $x$ for which $P(x)$ is false.

**Example** If we have $P(n) = (n^2 = 36)$ and domain of discourse $\mathbb{Z}$ then
$\forall n$ $P(n)$ is false
$\exists n$ $P(n)$ is true

### 4.2.1 Proof by existence and counter example

To prove $\forall x P(x)$ is true you have to prove $P(x)$ is true for all $x$ in the domain of discourse. Techniques such as proof by induction and contradiction can be used for this.

To prove that $\forall x P(x)$ is false you just have to find one value of $x$ in the domain of discourse for which $P(x)$ is false. This is called a proof by *counter example*.

To prove that $\exists x P(x)$ is true you just have to find one value of $x$ in the domain of discourse for which $P(x)$ is true. This is called a proof by *existence*.

To prove $\exists x P(x)$ is false you have to prove $P(x)$ is false for all $x$ in the domain of discourse. Again, techniques such as induction and contradiction can be used.

**Examples** In the following the domain of discourse is $\mathbb{Z}$.
(i) $\exists n(n^2 = 16)$ is true, proof by existence $n = 4$, $4^2 = 16$ is true.
(ii) $\forall n(n^2 = 4)$ is false, proof by counterexample $n = 1$, $1^2 \neq 4$.
(iii) $\exists n(n^2 = 3)$ is false (but it would be true if the domain of discourse were $\mathbb{R}$). This can be proved by contradiction using properties of division by primes.

**Exercises** What is the truth value of the following, given the domain of discourse is $\mathbb{Z}$.
1. $\forall n((n \text{ is divisible by } 4) \Rightarrow (n \text{ is even})$
2. $\exists n(n^2 = 4)$
3. $\forall n(n^2 > 0)$
4. $\exists n(integer(n) \wedge (n^2 = 5))$
5. $\exists y \forall x(x + y = 0)$
6. $\forall x \exists y(x + y = 0)$

## 4.3 Multiple parameters

What about statements with more than one variable?

"Any dog on the escalator must be carried by somebody"

$\forall x \exists y((on\_escalator(x) \wedge dog(x)) \Rightarrow (person(y) \wedge carried\_by(x, y)))$

Predicates can involve more than one object. For example, the property $y = x^2$. In this case we write $P(x, y)$. We can then have a quantifier for each object.

$\forall x \exists y$ $P(x, y)$ $\quad \exists y \forall x(x^2 = y)$ $\quad \exists y \forall x(\text{integer}(x) \Rightarrow (\text{integer}(y) \wedge (x^2 = y)))$
$\exists y \forall x$ $P(x, y)$ $\quad \forall x \exists y(x^2 = y)$ $\quad \forall x \exists y(\text{integer}(x) \Rightarrow (\text{integer}(y) \wedge (x^2 = y)))$

are the propositions

$$\text{for all } x \text{ there exists some } y \text{ such that, } y = x^2$$
$$\text{there exists some } y \text{ such that for all } x, \ y = x^2$$

If $x, y$ are integers then the first proposition is true, the second is false.

**Exercise** Decide whether each of the propositions $\forall y \exists x \ P(x, y)$ and $\exists y \exists x \ P(x, y)$ is true or false when $x, y$ are integers.

We can also form the negation of predicates, if $\neg \forall x \ P(x)$ is true exactly when $\forall x \ P(x)$ is false. In fact we have

$$\neg \forall x \ P(x) = \exists x \ \neg P(x)$$
$$\neg \exists x \ P(x) = \forall x \ \neg P(x).$$

**Exercises** Assume the domain of people, and that $L(x, y)$ means "$x$ likes $y$", $F(x)$ means "$x$ can speak French", and $J(x)$ means "$x$ knows Java". Write formulas for:
1. Some people don't like Fred.
2. There is a person who can speak French and knows Java.
3. Some people can speak French but don't know Java.
4. Everyone can speak French or knows Java.
5. No one can speak French or knows Java.
6. Everyone likes everyone else and themselves.
7. Some people like everyone except themselves.

## 4.4 n-ary Predicates

Predicates were used in CS1860 to define sets and relations were discussed in detail. Predicates in CS1860 usually had one variable and relations had two.

For example: $odd(x)$ (meaning that $x$ is an odd number) is a predicate; $less(x, y)$ (meaning that $x < y$) is a relation.

Both words are often used in a wider sense, and then the difference between them disappears. For example, $parents(x, y, z)$ (meaning that $x$ is $z$'s father and $y$ is $z$'s mother) can be referred to as a ternary predicate or a ternary relation.

Generally, an *n-ary predicate* (also referred to as an $n$-ary relation), where $n$ is a positive integer, is a property of $n$-tuples $(x_1, \ldots, x_n)$.

We use the terms "unary" for a predicate with 1 variable, "binary" for a predicate with 2 variables, and "ternary" for a predicate with 3 variables.

An *atomic sentence* in predicate logic has the form $predicate\_name(a_1, a_2, \ldots, a_n)$. The atomic sentence may contain variables, so that if precise objects are supplied for the variables the sentence becomes a proposition. For example, the atomic sentence "$likes(x, football)$" becomes a proposition for a particular $x$, such as "$likes(pete, football)$".

Names such as "pete" used to refer to particular values of variables are *constants*.

### Summary on predicates

We can use predicate symbols such as $P$, $Q$, $R$ to represent predicates in formulas. These symbols are predicate names.
• 1 argument: $P(x)$ is a unary predicate or property e.g., even($n$), friendly($x$)

- 2 arguments: $P(x, y)$ is a binary predicate or relation e.g., likes$(x, y)$, divisible$(n, 4)$
- 3 arguments: $P(x, y, z)$ is a ternary predicate e.g., parents$(x, y, z)$
- $n$ arguments: $P(a_1, a_2, \ldots, a_n)$ is an $n$-ary predicate, with $n$ arguments

## 4.5 Well formed formulae (wff)

We can use the logical symbols of propositional calculus to build logical expressions from predicates.

By convention, variables are written using $x$, $y$, $z$ and constants are written $a, b, c, \ldots$ We can also use predicate symbols such as $P, Q, R$. For example:

$\forall x(P(x) \Rightarrow \exists y Q(x, y))$
$\exists x(P(x) \wedge Q(x, c))$

The formal definition of a *well formed formula* (wff) is by structural induction (see Section 3.3 of Rosen) but for this course it is sufficient to think of a wff as any sentence constructed using predicates, variables, quantifiers and the logical connectives, $\wedge$, $\vee$, $\Rightarrow$ and $\neg$.

Wff form the basis of *predicate calculus*, the study of equality of logical expressions. We consider only first order predicate calculus in which the quantifiers can only range over variables, not functions.

We have the following logical equivalences for wwfs $A$ and $B$.

**Re-naming:**

$\forall x A(x) \Leftrightarrow \forall y A(y)$
$\exists x A(x) \Leftrightarrow \exists y A(y)$

**Negation:**

$\neg\forall x A(x) \Leftrightarrow \exists x(\neg A(x))$
$\neg\exists x A(x) \Leftrightarrow \forall x(\neg A(x))$

**Distributive laws:**

$\forall x(A(x) \wedge B(x)) \Leftrightarrow \forall x A(x) \wedge \forall x B(x)$
$\exists x(A(x) \vee B(x)) \Leftrightarrow \exists x A(x) \vee \exists x B(x))$

Convention: $\forall$ and $\exists$ have higher precedence than $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$.

**Exercises** Show that
$\forall x(P(x) \wedge \neg Q(x)) \Leftrightarrow (\forall x P(x) \wedge \neg\exists y Q(y))$
$\exists x(P(x) \Rightarrow Q(x)) \Leftrightarrow (\forall y P(y) \Rightarrow \exists x Q(x))$

## 4.6 Interpretations

A formula in propositional logic has a truth value associated with each possible truth value, T or F, of each of its propositional variables. For example, $P \Rightarrow Q$ is true for $P = T, Q = T$, but false for $P = T, Q = F$.

A mapping from each of the variables to T or F is called an *interpretation*, and gives a meaning to the formula.

Similarly formulae in predicate logic have a truth value when we have given values to all the variables. However, the process of giving an interpretation to a predicate logic

formula is more complicated.

### 4.6.1 Interpretations in predicate logic

Consider an interpretation of the formula $\forall x \forall y(P(x, y))$ where the domain of discourse for both $x$ and $y$ is $\{sue, ann, john, bill\}$, and where $P$ is the relation $likes$ and we have

$$likes = \{(sue, john), (john, sue), (sue, ann), (bill, ann), (sue, bill), (ann, sue)\}$$

In logic we often use the terminology

$$likes(sue, john), likes(john, sue), likes(sue, ann),$$

$$likes(bill, ann), likes(sue, bill), likes(ann, sue)$$

to specify the binary relations $likes$ and the convention is that $likes(x, y) = T$ in the above cases and $likes(x, y) = F$ for all other combinations of $x$ and $y$.

The above formula is *false* under this interpretation because we can find a counter-example, e.g., $x = ann, y = bill$

**Exercise** What is the truth value of $\exists x \forall y(((x \neq y) \Rightarrow P(x, y)) \wedge \neg P(x, x))$ under the above interpretation?

**Exercises** Consider the interpretation: Domain = {margaret, george, tony, harriet}
Predicates:
$P$ = leader, and is true for {george, tony, michael}
$Q$ = likes, and is true for {(george, tony), (tony, george), (michael, tony)}

Find the truth values under this interpretation of:

1. $\exists x \exists y(Q(x, y) \wedge Q(y, x))$
2. $\forall x(P(x) \vee \exists y Q(x, y))$
3. $\exists x \forall y(\neg Q(x, y) \wedge \neg Q(y, x))$
4. $\forall x(P(x) \Rightarrow \exists y Q(x, y))$

### 4.6.2 Semantic entailment

**Definitions** An interpretation which makes a formula $A$ true is a *model* for $A$. A formula which has at least one model is said to be *consistent* or *satisfiable*. A formula which has no models is said to be *inconsistent* or *unsatisfiable*. A formula which is true for all interpretations is said to be *valid*. A formula which is neither inconsistent nor valid is said to be *contingent*.

A formula $A$ *semantically entails* formula $B$ if and only if every model for $A$ is also a model for $B$. That is, any interpretation which makes $A$ true also makes $B$ true.

We write:

$$A \models B$$

We can also say that $A$ logically implies $B$, or $B$ is a logical consequence of $A$.

## 4.7 Inference and proof

A proof is a logical argument which ends with the conclusion that some proposition is true. We state the assumptions, or hypotheses, and these form the axioms on which the proof is based. We then use rules to deduce consequences of these axioms, until we get the required proposition.

In the last section we discussed the idea of the logical consequence of a formula in logic: $A \models B$ means any interpretation which makes formula $A$ true also makes formula $B$ true. We can also say that $A$ *logically implies* $B$, or $B$ is a *logical consequence* of $A$.

We can apply the same idea to a set of formulae: $S \models A$ means that any interpretation which makes all the formulae in set $S$ true also makes $A$ true. That is: $A$ is *semantically entailed* by the set $S$, $A$ is a *logical consequence* of $S$.

### 4.7.1 Demonstrating logical consequences

In propositional logic, to demonstrate that $S \models U$, we can construct the truth tables.

Write the truth table with a column for each of the propositions in $S$ and a column for $U$ and check that that in *every* row in which all of the propositions in $S$ are true, $U$ is also true.

**Example** Show that $\{P, Q, Q \Rightarrow (R \vee U), \neg R\} \models U$.

| P | Q | R | U | R∨U | Q⇒(R∨ U) | ¬R | |
|---|---|---|---|---|---|---|---|
| T | T | T | T | T | T | F | F |
| T | T | T | F | T | T | F | F |
| T | T | F | T | T | T | T | T |
| T | T | F | F | F | F | T | F |
| T | F | T | T | T | T | F | F |
| T | F | T | F | T | T | F | F |
| T | F | F | T | T | T | T | F |
| T | F | F | F | F | T | T | F |

The last column is the truth value of the conjunction of propositions in the set $S$.

Note, we only need the rows for which $P = T$ so we have left the other rows out. The row in which the LHS propositions are all true is row 3, and for this row $U = T$, as required.

**Exercise** Use truth tables to show that $\{Q, P \Rightarrow Q, Q \Rightarrow R\} \models (R \wedge Q)$

### 4.7.2 Proof using logical inference

Using truth tables for checking large semantic entailments is cumbersome, and for predicate logic it is not possible if the domain of discourse is infinite.

It is often possible to prove semantic entailments by applying inference rules. You can think of truth tables as showing something is true by looking at every element, while applying inference rules is using mathematical reasoning to give a proof.

If $S$ is a set of formulae and $A$ is a single formula, then $A$ *can be proved from* $S$ if $A$ can be obtained from $S$ by application of sound inference rules.

We write $S \vdash A$

We can also say that $A$ *can be derived* from $S$, or that $S$ *syntactically entails* $A$.

We now describe the inference rules.

### 4.7.3 Modus ponens

If we know that a property $P$ implies a property $Q$, and that $P$ is true then we can deduce that $Q$ is true.

For example,

- ◇ If it is true that when I am at the seaside then I am happy,
- ◇ if it is true that I am at the seaside,

then you can deduce that I am happy.

*Modus ponens* is the inference rule: from (if $P$ then $Q$) and $P$, deduce $Q$.

That is, from hypotheses $P \Rightarrow Q$ and $P$, we can infer $Q$.

We write either $\dfrac{P, P \Rightarrow Q}{Q}$ or $\{P, P \Rightarrow Q\} \vdash Q$

In addition to single propositions $P$ and $Q$, we can have formulae $A$ and $B$

$$\frac{A, A \Rightarrow B}{B}$$

**Example** Consider the propositions

$P$ = Interest rates increase $Q$ = Mortgage rates increase $R$ = House prices fall

and the hypotheses $\{P \Rightarrow Q, Q \Rightarrow R, P\}$

We can use modus ponens to show that $\{P \Rightarrow Q, Q \Rightarrow R, P\} \vdash R$

$$\frac{\dfrac{P, P \Rightarrow Q}{Q} \qquad Q \Rightarrow R}{R}$$

We can also write the proof using the $\vdash$ notation:

$$\{P, P \Rightarrow Q\} \vdash Q, \qquad \{Q, Q \Rightarrow R\} \vdash R$$

### 4.7.4 Soundness

We only want to use an inference rule if things proved using it are correct in the sense the if $A$ is proved from $S$ then $S \models A$. If any formula $A$ which can be derived from a set $S$ of other formulae using one or more applications of a rule of inference is also a logical consequence of $S$, then the rule of inference is said to be *sound*.

In other words, anything that we derive from $S$ using that rule of inference will be true for all interpretations which make all the formulae in $S$ true (all models of $S$).

**Definition** A *sound inference rule* $R$ is one for which If $S \vdash A$ (using $R$) then $S \models A$.

We can look at the truth table for $P \Rightarrow Q$

| P | Q | P⇒ Q |
|---|---|---|
| T | T | T |
| T | F | F |
| F | T | T |
| F | F | T |

We can see that if $P = T$ and $(P \Rightarrow Q) = T$ then only row 1 of the table is relevant and in this row $Q = T$.

We have shown above that $\{P \Rightarrow Q, Q \Rightarrow R, P\} \vdash R$ using modus ponens. It can be checked using truth tables that $\{P \Rightarrow Q, Q \Rightarrow R, P\} \models R$.

### 4.7.5 Some inference rules

There are lots of sound inference rules.

$$\frac{A}{A \vee B}$$

$$\frac{A \wedge B}{A}$$

$$\frac{A, B}{A \wedge B}$$

$$\frac{A, A \Rightarrow B}{B}$$

$$\frac{\neg B, A \Rightarrow B}{\neg A}$$

$$\frac{A \Rightarrow B, B \Rightarrow C}{A \Rightarrow C}$$

$$\frac{A \vee B, \neg A}{B}$$

The soundness of each of these inference rules can be shown using truth tables.

### 4.7.6 Direct proofs with inferences

We can derive a result using any combination of sound inference rules.

**Example** If I have a cold, I sneeze. If I sneeze, either I have hay-fever or I should stay in bed. I have a cold and I don't have hay-fever. So I should stay in bed.

Model this with $P =$ "I have a cold", $Q =$ "I sneeze", $R =$ "I have hay fever", $U =$ "I should stay in bed"

Show that $\{P, P \Rightarrow Q, Q \Rightarrow (R \vee U), \neg R\} \vdash U$

$$\dfrac{\dfrac{\dfrac{P, P \Rightarrow Q}{Q} \qquad Q \Rightarrow (R \vee U)}{R \vee U} \qquad \neg R}{U}$$

**Exercise** Using direct proof by inferences, show that $\{P \Rightarrow (Q \Rightarrow R), P \wedge Q\} \vdash R$

### 4.7.7 Proof by contradiction

If we want to show that

$$S \models A$$

it is sufficient to show that

$$S \cup \{\neg A\} \models \bot$$

where $\bot$ is a symbol which in this case denotes the truth value FALSE.

If $S \cup \{\neg A\} \models \bot$ then this means that no row of the truth table for all the propositions in $S \cup \{\neg A\}$ has all Ts. This means that if a row has Ts in all the columns for $S$ then $\neg A$ must be F, and hence $A$ is T. So $S \models A$.

A similar argument shows that the converse is also true, if $S \models A$ then $S \cup \{\neg A\} \models \bot$. So if $S \cup \{\neg A\} \not\models \bot$ then $S \not\models A$.

So, to prove a result by contradiction, add the negation of what you want to prove to the hypotheses, and derive an expression which must be false.

We express this as the inference rule

$$\frac{A, \neg A}{\bot}$$

**Example** We prove $\{P, P \Rightarrow Q, Q \Rightarrow R\} \models R \wedge Q$ by contradiction. Assuming $\neg(R \wedge Q)$, we show $\{P, P \Rightarrow Q, Q \Rightarrow R, \neg(R \wedge Q)\} \vdash \bot$.

$$\dfrac{\dfrac{\dfrac{P, P \Rightarrow Q}{Q} \quad \dfrac{\neg(R \wedge Q)}{\neg R \vee \neg Q}}{\neg R} \qquad \dfrac{\dfrac{P, P \Rightarrow Q}{Q} \quad Q \Rightarrow R}{R}}{\bot}$$

## 4.8 Practice exercises

1. Let $n$ be an integer and let $P(n)$ be the predicate '$n^2 - 2n + 1 = 0$'. What are the truth values of
   a. $\forall n P(n)$
   b. $\exists n P(n)$.

2. Write down, in predicate form, the negation of the following propositions, where $x, y$ are real numbers.
   a. $\exists x \forall y (y$ exactly divides $x)$
   b. $\forall x \exists y (x - y = 17)$.

3. Write the following statements in predicate form.
   a. All dogs bark
   b. There does not exist an $x$ such that $x = 2x$
   c. For each $y$, $y^2$ is greater than $y$
   d. No computer science student is not mathematically literate

4. Is the predicate in Question 2(b) above true? (Justify your answer.)

5. Use truth tables to show that $\{Q \wedge (P \vee R), Q \Rightarrow R\} \models R$.

6. Use inference rules to show that $\{Q \wedge (P \vee R), Q \Rightarrow R\} \vdash R$.

# Chapter 5

# Logic circuits

Most of the material in this chapter is covered in Chapter 9 of Rosen.

In the rest of this course we are going to look at four different views of computing machines. The lowest level is as an electrical circuit of on/off switches (switching circuits). The next level is as a network of logic gates (logic circuits) and the highest level is from the perspective of machine instructions (assembler level). We also look at machines from the perspective of transitioning from one state to another in response to inputs (automata theory). We begin with the logic circuits.

We are used to two types of data. Discrete values are distinct, like the integers. Analogue data is continuous, given any value $x$ and any distance $\delta > 0$ there is another value whose distance from $x$ is less than $\delta$.

Discrete data can be modelled using the integers and is often referred to as digital information. Analogue data can be expressed to arbitrary levels of precision.

Examples of digital information are data sets such as the binary numbers $\{0, 1\}$, Boolean truth values. $\{True, False\}$, a set of colours $\{Red, Green, Blue\}$.

Examples of analogue information are continuously variable physical quantities such as speed as measured by a speedometer, temperature as measured by a thermometer, an interval of time as measured by a grandfather clock, ...

Most computers work in binary $\{0, 1\}$. They break an analogue quantity, a voltage, into a digital quantity by taking any voltage around 0V to be equivalent to 0 and any voltage around 5V to be equivalent to 1.

The binary code can then be used to represent any discrete quantity.

## 5.1 Processing information

Computer instructions are built from elementary logic operations. Binary information, in particular truth values, are represented as $\{0, 1\}$ with, in this course, 1 representing true and 0 representing false.

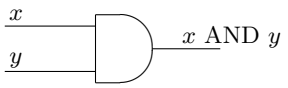
We define logic gates for several logical operators. The gates have one or two inputs and an output. Later we shall consider gates with more than two inputs.

For logic circuits is it common to use mnemonic names for the logical operators rather than the mathematical symbols, $\wedge$ (AND), $\vee$ (OR), and $\neg$ (NOT). You need to be able to use both notations. When doing assignments and exams use the same notation as is used in the question you are answering.

An AND-gate takes two inputs and outputs the value of their conjunction.

| $x$ | $y$ | $x$ AND $y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

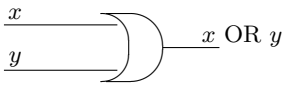
As a logic gate:




An OR-gate takes two inputs and outputs the value of their disjunction.

| $x$ | $y$ | $x$ OR $y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

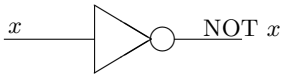
As a logic gate:




A NOT-gate takes one input and outputs its negation.

$$\text{NOT } x \quad = \quad \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{if } x = 1 \end{cases}$$

As a logic gate:



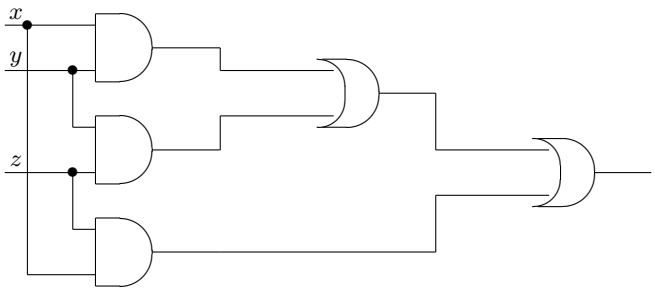

**Example** Majority Voting

We can use logic gates to build a system which computes the majority vote among three voters who have a choice of either either yes (1) or no (0).

If two or more people say yes then the decision should be yes, otherwise the decision should be no. If $x$, $y$ and $z$ correspond to the votes and $r$ to the result we have

$$r = (x \text{ AND } y) \text{ OR } (y \text{ AND } z) \text{ OR } (x \text{ AND } z)$$

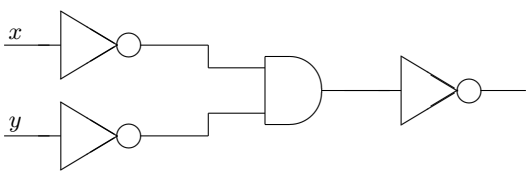
A logic circuit which computes this expression is



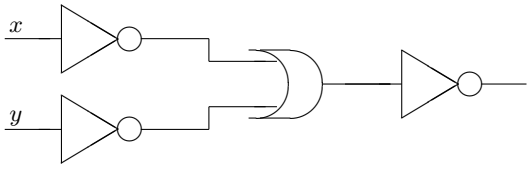

## 5.1.1 Operator redundancy

Since any logic expression has a DNF, it can be built using the AND, OR and NOT operators. Recall, for example, that $(x \Rightarrow y) = (\text{NOT } x) \vee\ y$.

In fact, you only need NOT and either AND or OR. It is easy to check using truth tables that $x$ OR $y = \text{NOT}(\text{NOT}x \text{ AND NOT}y)$, this is one of De Morgan's laws. So we could replace the $OR$ gates with




Similarly we have that the following circuit uses only NOT and OR gates to compute AND.




This means that we can build any logic circuit, any computer, using just AND and NOT gates, or indeed just OR and NOT gates.

### 5.1.2 NAND, NOR and XOR

Perhaps more surprisingly there is a logical operator which on its own can generate all the others. In fact there are two such operators, and they are as simple as AND.

We define NAND to be not AND and we define NOR to be not OR. So

$$x \text{ NAND } y = \text{ NOT } (x \text{ AND } y)$$

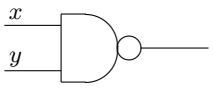
$$x \text{ NOR } y = \text{ NOT } (x \text{ OR } y)$$

The truth tables for these operators are

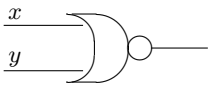
| $x$ | $y$ | $x$ NAND $y$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| $x$ | $y$ | $x$ NOR $y$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

We drawn NAND and NOR gates in the same way as the AND and OR gates but with a small circle on the output.




NAND gate




NOR gate

The following logical equivalences hold

$$\begin{aligned} \text{NOT } x &= x \text{ NAND } x \\ x \text{ AND } y &= \text{ NOT } (x \text{ NAND } y) \\ x \text{ OR } y &= (\text{ NOT } x) \text{ NAND } (\text{ NOT } y) \end{aligned}$$

$$\begin{aligned} \text{NOT } x &= x \text{ NOR } x \\ x \text{ OR } y &= \text{ NOT } (x \text{ NOR } y) \\ x \text{ AND } y &= (\text{ NOT } x) \text{ NOR } (\text{ NOT } y) \end{aligned}$$

Thus we see that any logical expression can be written using just NAND operators. Indeed, computers could be built solely of NAND gates.

Notice that OR has a slightly different meaning to the common English usage. When someone says

I will go to the shops or to the gym

we normally assume that they will do one or the other but not both. In Mathematics OR allows both of its operands to be true, to emphasise this we sometimes call it *inclusive* OR. There is another form of OR, called *exclusive* OR and written XOR, which has the normal English meaning, i.e. one of its arguments is true but not both of them.

| $x$ | $y$ | $x$ XOR $y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

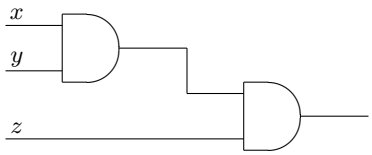
### 5.1.3 Multi-input gates

AND and OR can be genralized too any number of inputs. Diagrammatically, we simply add extra input lines:

multiple OR multiple AND

The meaning of a multiple input gates is an extension of original meaning. For example, the output of multiple AND is 0 if any of the inputs are zero, otherwise it is 1.

Devices implementing multiple gates exist, but they can also be easily constructed from two input devices. For example, a 3-input AND gate can be constructed as




## 5.2 Algebraic notation

We have already used two different notations for the logical operators. Now we consider the algebraic notation which is more compact.

| | | | |
|---|---|---|---|
| $xy$ | means | $x$ AND $y$ | $x \wedge y$ |
| $x + y$ | means | $x$ OR $y$ | $x \vee y$ |
| $\overline{x}$ | means | NOT $x$ | $\overline{x}$ |
| $x \oplus y$ | means | $x$ XOR $y$ | |

**Examples** We give some examples using both the logic and algebraic notations.

$$xy + yz + zx \text{ is equivalent to } (x \text{ AND } y) \text{ OR } (y \text{ AND } z) \text{ OR } (x \text{ AND } z)$$

$$x\,(y + z) \text{ is equivalent to } x \text{ AND } (y \text{ OR } z)$$

$$x\,(\overline{y + z}) \text{ is equivalent to } \begin{cases} x \text{ AND NOT } (y \text{ OR } z) \\ \text{and also} \\ x \text{ AND } (y \text{ NOR } z) \end{cases}$$

The following is a list of logical equivalences in algebraic notation.

$$\begin{aligned} \overline{\overline{x}} &= x & x\overline{x} &= 0 \\ xx &= x & x + \overline{x} &= 1 \\ x + x &= x & x0 &= 0 \\ xy &= yx & x1 &= x \\ x + y &= y + x & x + 0 &= x \\ x(y + z) &= xy + xz & x + 1 &= 1 \\ x + (y + z) &= (x + y) + z & x(yz) &= (xy)z \end{aligned}$$

De Morgan's Laws:

$$\begin{aligned} \overline{xy} &= \overline{x} + \overline{y} \\ \overline{x + y} &= \overline{x}\,\overline{y} \end{aligned}$$

Note, be careful. In general, $\overline{x}\,\overline{y} \neq \overline{xy}$.

## 5.3 Building logic circuits

For certain types of logical expression (DNFs, see below) it is straightforward to build a corresponding logic circuit using NOT gates, multi-input AND gates and a multi-input OR gate.

**Exercise** Using NOT gates, and multi-input AND, and OR logic gates draw logic circuits representing the formulae

- $\diamond$ $xy + \overline{x}\,\overline{y}$
- $\diamond$ $\overline{x}\,\overline{y}z + \overline{x}y + x\overline{z}$

Now draw equivalent circuits which use only binary input AND and OR gates.

Given a logical expression in any format, how do we construct a logic circuit which implements it? Furthermore, how do we construct a small such circuit? We want an automated process for this, so we need an algorithm which takes ANY logic expression and returns an efficient corresponding logic circuit.

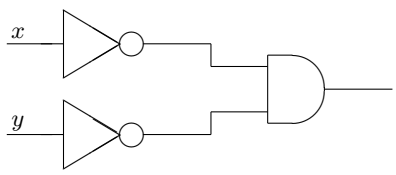
An algebraic expression is in disjunctive normal form if and only if it is a sum of products of variables and their negations. These products are often called the *addends* of the DNF. We can turn a DNF into a circuit of multi-input logic gates easily. We have one multi-input AND gate for each conjunction in the DNF and put the outputs of these AND gates into a single multi-input OR gate. This is gives us an algorithm for constructing a logic circuit for any logical expression:

1. construct the truth table
2. construct a DNF from the truth table
3. create a NOT gate for each variable which appears negated in the DNF
4. create a multi-input AND gate for each conjunction in the DNF with inputs $x$ or $\overline{x}$ depending on whether $x$ or $\overline{x}$ is in the conjunction
5. if there is more than one AND gate create a multi-input OR gate whose inputs are the outputs of all the AND gates

**Example** Consider $(x \Rightarrow y) \wedge \neg y$. Its truth table is

| $x$ | $y$ | $x \Rightarrow y$ | $\neg y$ | $(x \Rightarrow y) \wedge \neg y$ |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 |

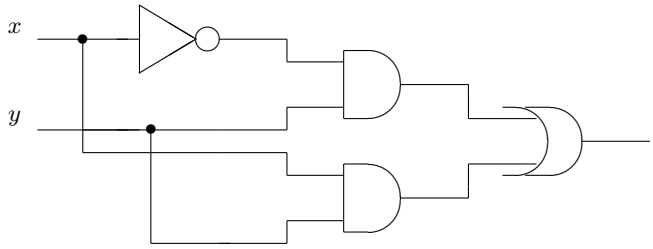
giving the DNF $\overline{x}\,\overline{y}$ and logic circuit



**Example** Consider $(x \Rightarrow y) \wedge y$. Its truth table is

| $x$ | $y$ | $x \Rightarrow y$ | $(x \Rightarrow y) \wedge y$ |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |

giving the DNF $\overline{x}\,y + x\,y$ and logic circuit




### 5.3.1 Minterms

A *minterm* is a product which contains exactly one instance of each input variable or its negation. For three inputs $x$, $y$, $z$ the following are minterms $xyz$, $\overline{x}yz$, and $\overline{x}y\overline{z}$ but $\overline{xy}z$ is not a minterm.

In algebraic notation a DNF is a sum of products and the minterms are the conjunctions in the DNF.

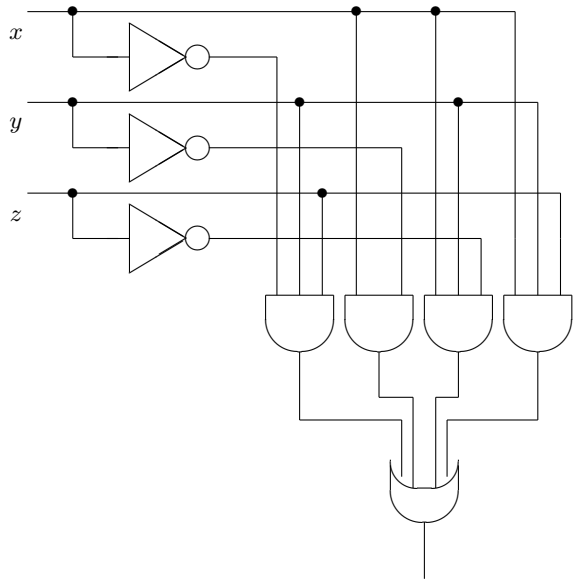
In general, if there are $n$ inputs, then there are $2^n$ minterms.

**Example** Recall the Majority Voting function $r$ whose truth table is

| $x$ | $y$ | $z$ | $r$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

We can compute a DNF for $r$ from this table by looking at the rows in which $r$ has value 1. In algebraic notation we get

$$\overline{x}yz + x\overline{y}z + xy\overline{z} + xyz$$




Of course this is not the most efficient solution. We need to enhance the algorithm so that it can produces circuits with fewer gates. We want to add an additional step in which we construct a smaller DFA before building the AND gates.

### 5.3.2 Constructing a smaller DNF

When we first considered the majority voting problem we found the expression

$$yz + xz + xy,$$

We know that $\overline{r} + r = 1$ and $r1 = r$ for any logical expression $r$ so

$$xy\overline{z} + xyz \;=\; xy(\overline{z} + z) \;=\; xy$$

We also know that $r = r + r$ so we can introduce two copies of $xyz$ and use them with the other two minterms

$$\overline{x}yz + x\overline{y}z + xy\overline{z} + xyz \;=\; \overline{x}yz + x\overline{y}z + xy\overline{z} + xyz + xyz + xyz \;=\; yz + xz + xy$$

We generalise this observation to get a procedure for reducing the size of a DNF: combine the minterms (and add repeated terms if necessary) to obtain terms of the form

$$\ldots(x + \overline{x})\ldots$$

From this get a smaller equivalent DNF.

Formally, take an existing DNF $\alpha_1 + \ldots \alpha_d$ and write down a new DNF as follows.

1. Consider each addend (minterm) $\alpha_i$ in the DNF in turn. If there is another addend $\alpha_j$ which differs from $\alpha_i$ in just one variable then add the new addend to the new DNF, otherwise add $\alpha_i$ to the new DNF.

2. Remove any addend from the new DNF which also has a subproduct in the new DNF.

**Example** Suppose that $r$ is specified by the following table.

| $x$ | $y$ | $z$ | $r$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

From this we read off the DNF

$$\overline{x}\,\overline{y}z + \overline{x}y\overline{z} + x\overline{y}\,\overline{z} + x\overline{y}z + xyz$$

We match $\overline{x}\,\overline{y}z$ to $x\,\overline{y}z$ and get $\overline{y}z$. The product $\overline{x}y\overline{z}$ does not match any others so is used as it is. The product $x\overline{y}\,\overline{z}$ matches $x\overline{y}\,z$ to give $x\overline{y}$. Finally, $x\overline{y}z$ is matched to $\overline{x}\,\overline{y}z$, generating $\overline{y}z$ and $xyz$ is matched to $x\overline{y}z$ creating $xz$. This gives the DNF

$$\overline{y}z + \overline{x}y\overline{z} + x\overline{y} + \overline{y}z + xz$$

from which we remove one of the repeated addends $\overline{y}z$ to get $\overline{y}z + \overline{x}y\overline{z} + x\overline{y} + xz$.

**Example** Change the previous example slightly, inverting the value of two rows.

| $x$ | $y$ | $z$ | $r$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

From this we read off the DNF

$$\overline{x}\,\overline{y}z + \overline{x}y\overline{z} + \overline{x}yz + x\overline{y}\,\overline{z} + x\overline{y}z + xy\overline{z} + xyz$$

At the first step, the product $\overline{x}y\overline{z}$ now matches $\overline{x}yz$ and generates $\overline{x}y$, and the product $xy\overline{z}$ can match $\overline{x}y\overline{z}$ to generate $y\overline{z}$. After removing repeats we get a DNF

$$\overline{y}z + \overline{x}y + x\overline{y} + y\overline{z} + xz$$

(note, this process is not unique and different DNFs can be found.)

This has given us a smaller DNF but not the smallest possible. The issue is that there was a larger factorisation that we could have performed on the original DNF. We have applied the rule $\overline{r} + r = 1$ but it is also true that $\overline{r}\overline{s} + r\overline{s} + \overline{r}s + rs = 1$.

In general, for any $n$ the DNF (sum) of all of the minterms in $n$ variables is 1 (true).

Using this observation on the DNF

$$\overline{x}\,\overline{y}z + \overline{x}y\overline{z} + \overline{x}yz + x\overline{y}\,\overline{z} + x\overline{y}z + xy\overline{z} + xyz$$

we find that, for example,

$$x\overline{y}\,\overline{z} + x\overline{y}z + xy\overline{z} + xyz$$

matches $x$, so we can replace all four of these minterms with just $x$. Similarly, we can find four matches which generate $y$ and four which generate $z$, giving the minimal DNF

$$z + y + x$$

There is a graphical approach which can be used to identify factorisations involving the sums of all minterms. It works well for expressions of 3 or 4 variables, and reasonably well for expressions of upto about 6 variables. In this course we will just look at 2 and 3 variables.

**Exercises** Draw logic circuits for the following logical expressions.
1. $xy + x$, using any gate types
2. $x \Rightarrow y$, using AND, OR and NOT gates
3. $\overline{x}\ \overline{y} + x\overline{y} + xy$, using AND, OR and NOT gates
4. $x\,\overline{y}\,z + x\,\overline{y}\,\overline{z}$, using NOT, OR and multi-input AND gates
5. $\overline{xy}$, using only NAND gates
6. $x \Rightarrow y$, using only NAND gates

### 5.3.3 Karnaugh maps

For 2 and 3 variables we can write truth tables so one can spot the factorisations more easily. The minimisation process from the previous section can be described graphically using what are called Karnaugh maps. This method allows us to identify the 'collapsing' combinations. An alternative way of writing truth tables for small numbers of variables is to label the rows and columns with the values of the variables.

We begin with two variables. There are four possible choices for the values, $x = 0, 1$ and $y = 0, 1$. We write the values of $x$ on the rows, the values of $y$ on the columns, and in the corresponding cell we put the value of the expression for that value of $x$ and $y$. The top row corresponds to $x$ false, i.e. $\overline{x} = 1$, and the left column to $\overline{y} = 1$.

**Example** Consider the expression $r = xy + y$. The table for this is

| | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 0 | 1 |

equivalent to

| $x$ | $y$ | $xy$ | $xy + y$ |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 |

The rows are the values of $x$ and the columns are the values of $y$. As we did for DNF above, to find the minterms for which the expression evaluates to true we look at each table entry which has value 1. It can be helpful to label the rows and columns of the table with the variables and their negations, this makes reading off the minterms easy. It is important to use a common convention, the convention is that the values in the top row are computed with $x$ being false, and the values in the first column are computed with $\overline{y} = 1$, etc.

| | $\overline{y}$ | $y$ |
|---|---|---|
| $\overline{x}$ | 0 | **1** |
| $x$ | 0 | **1** |

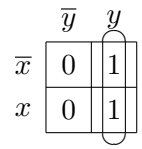
In this case we get the DNF $xy + \overline{x}y$.

Two 1s in a column indicate that the column element is in a minterm with both $\overline{x}$ and $x$. Two 1s in the first column mean that we have $\overline{x}\,\overline{y} + x\overline{y}$, which can be replaced by just

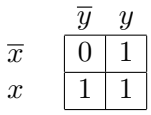
$\overline{y}$. Similarly two 1s in the second column give us just $y$, and two 1s in the rows give us $\overline{x}$ or $x$.

Graphically, we mark each simplification by drawing a loop around the appropriate 1s. If a 1 is not part of a row or column pair then a loop is drawn around it on its own. Once all the 1s are in at least one loop the minimal DNF is read off, one addend for each loop.

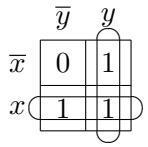
For the above example we have one loop, and the DNF is $y$ (the label of the column containing the loop).




**Example** Consider the expression $r = x + \overline{x}y$. The Karnaugh map for this is

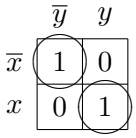



We can draw a loop around the right column. This still leaves one 1 uncovered, but we can draw another loop around the lower row. The 1s are all then covered and we can read off the DNF $x + y$.

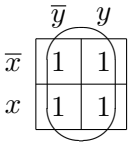



**Example** (a) For the Karnaugh map of $\overline{x}\,\overline{y} + xy$ we have no columns or rows, so the covering loops are as shown on the left below and the expression is already minimal.
(b) For the Karnaugh map with 1s in every position we have a large loop around the whole table and the DNF is 1.

(a)




(b)




**Exercise** Write out the Karnaugh map for $r = x + \overline{x}\,\overline{y}$ and use it to construct a minimised expression.

## 5.3.4 Karnaugh maps in three variables

Minimising an expression in only two variables is straightforward as there are only a few cases which can arise. We have descibed the case of two variables in detail as a precursor to the three variable case.

There are $2^3 = 8$ rows in the truth table for a 3-variable expression, so we need a Karnaugh map with 8 cells. We can draw this as a $2 \times 4$ table but we need to choose

carefully how the columns are labelled in order for adjacent 1s to correspond to identity factorisations. A labelling that works, and the one we use in this course is

| | $\overline{y}$ | $y$ | $y$ | $\overline{y}$ |
|---|---|---|---|---|
| $\overline{x}$ | | | | |
| $x$ | | | | |
| | $\overline{z}$ | $\overline{z}$ | $z$ | $z$ |

The table is thought of as self looping so the first and last columns are adjacent (imagine the table wrapped around a cylinder).

**Example** The following is the Karnaugh map for the majority voting expression

| | $\overline{y}$ | $y$ | $y$ | $\overline{y}$ |
|---|---|---|---|---|
| $\overline{x}$ | 0 | 0 | 1 | 0 |
| $x$ | 0 | 1 | 1 | 1 |
| | $\overline{z}$ | $\overline{z}$ | $z$ | $z$ |

We draw a loop around the middle pair of 1s in the $x$-row, they have the same $x$ and $y$ values but complementary $z$ values. This gives the expression $xy$. We can also draw a loop around the right-most pair of 1s, giving $xz$, and the third column, giving $yz$.

| | $\overline{y}$ | $y$ | $y$ | $\overline{y}$ |
|---|---|---|---|---|
| $\overline{x}$ | 0 | 0 | 1 | 0 |
| $x$ | 0 | 1 | 1 | 1 |
| | $\overline{z}$ | $\overline{z}$ | $z$ | $z$ |

We have now *covered* all the 1s in the table, and have the DNF $xy + xz + yz$.

To identify a factorisation of the form $\overline{r}\,\overline{s} + \overline{r}s + r\overline{s} + rs$ we draw loops around blocks of four adjacent 1s if possible (bigger blocks are better).

**Example** Consider $r = x\overline{z} + \overline{x}y\overline{z} + yz + x\overline{y}z$ whose Karnaugh map is:

| | $\overline{y}$ | $y$ | $y$ | $\overline{y}$ |
|---|---|---|---|---|
| $\overline{x}$ | 0 | 1 | 1 | 0 |
| $x$ | 1 | 1 | 1 | 1 |
| | $\overline{z}$ | $\overline{z}$ | $z$ | $z$ |

We can draw several size loops:
- 2 x 1 loops, e.g., $xz$
- 4 x 1 loops, e.g., $x$
- 2 x 2 loops, e.g., $y$

| | $\overline{y}$ | $y$ | $y$ | $\overline{y}$ |
|---|---|---|---|---|
| $\overline{x}$ | 0 | 1 | 1 | 0 |
| $x$ | 1 | 1 | 1 | 1 |
| | $\overline{z}$ | $\overline{z}$ | $z$ | $z$ |

The $4 \times 1$ and $2 \times 2$ loops cover all the 1s, so the corresponding terms $x$ and $y$ are sufficient, and hence $r = x + y$.

**Example** Consider $r = \overline{x}\,\overline{y}\,\overline{z} + \overline{x}\,\overline{y}z + xy\overline{z}$ whose Karnaugh map is:

| | $\overline{y}$ | $y$ | $y$ | $\overline{y}$ |
|---|---|---|---|---|
| $\overline{x}$ | 1 | 0 | 0 | 1 |
| $x$ | 0 | 1 | 0 | 0 |
| | $\overline{z}$ | $\overline{z}$ | $z$ | $z$ |

or equivalently

| | $y$ | $y$ | $\overline{y}$ | $\overline{y}$ |
|---|---|---|---|---|
| $\overline{x}$ | 0 | 0 | 1 | 1 |
| $x$ | 1 | 0 | 0 | 0 |
| | $\overline{z}$ | $z$ | $z$ | $\overline{z}$ |

There is a $2 \times 1$ pattern in the first row. (Remember the first and last columns are considered to be adjacent, you can imagine that the Karnaugh map has been rewritten as shown on the right above.) This leaves the single 1 in the second row which cannot be covered by a $2 \times 1$ loop thus we have to use a singleton cover. Then we get the following cover, which generates the DNF $\overline{x}\ \overline{y} + xy\overline{z}$.

| | $\overline{y}$ | $y$ | $y$ | $\overline{y}$ |
|---|---|---|---|---|
| $\overline{x}$ | 1 | 0 | 0 | 1 |
| $x$ | 0 | 1 | 0 | 0 |
| | $\overline{z}$ | $\overline{z}$ | $z$ | $z$ |

## 5.3.5 Fundamental patterns for 3 variables

There are $2^8 = 256$ Karnaugh maps with three variables, so although they could all be written out it is a large set to remember. However, it is useful to remember the basic patterns of loops that can occur, and their corresponding logic expression. In this section we give examples of these patterns. Blocks of 4 give a product of one variable, blocks of 2 give a product with two variables and singletons give a product with three variables, so to get the smallest addends draw the largest possible loops.

| | $\overline{y}$ | $y$ | $y$ | $\overline{y}$ |
|---|---|---|---|---|
| $\overline{x}$ | 1 | 1 | 1 | 1 |
| $x$ | 1 | 1 | 1 | 1 |
| | $\overline{z}$ | $\overline{z}$ | $z$ | $z$ |

addend = 1

| | $\overline{y}$ | $y$ | $y$ | $\overline{y}$ |
|---|---|---|---|---|
| $\overline{x}$ | 1 | 1 | 1 | 1 |
| $x$ | | | | |
| | $\overline{z}$ | $\overline{z}$ | $z$ | $z$ |

addend = $\overline{x}$

| | $\overline{y}$ | $y$ | $y$ | $\overline{y}$ |
|---|---|---|---|---|
| $\overline{x}$ | | | 1 | 1 |
| $x$ | | | 1 | 1 |
| | $\overline{z}$ | $\overline{z}$ | $z$ | $z$ |

addend = $z$

| | $\overline{y}$ | $y$ | $y$ | $\overline{y}$ |
|---|---|---|---|---|
| $\overline{x}$ | 1 | | | 1 |
| $x$ | 1 | | | 1 |
| | $\overline{z}$ | $\overline{z}$ | $z$ | $z$ |

addend = $\overline{y}$

| | $\overline{y}$ | $y$ | $y$ | $\overline{y}$ |
|---|---|---|---|---|
| $\overline{x}$ | | 1 | | |
| $x$ | | 1 | | |
| | $\overline{z}$ | $\overline{z}$ | $z$ | $z$ |

addend = $y\overline{z}$

| | $\overline{y}$ | $y$ | $y$ | $\overline{y}$ |
|---|---|---|---|---|
| $\overline{x}$ | | | 1 | 1 |
| $x$ | | | | |
| | $\overline{z}$ | $\overline{z}$ | $z$ | $z$ |

addend = $\overline{x}z$

| | $\overline{y}$ | $y$ | $y$ | $\overline{y}$ |
|---|---|---|---|---|
| $\overline{x}$ | | | | |
| $x$ | | | 1 | |
| | $\overline{z}$ | $\overline{z}$ | $z$ | $z$ |

addend = $xyz$

| | $\overline{y}$ | $y$ | $y$ | $\overline{y}$ |
|---|---|---|---|---|
| $\overline{x}$ | | | | |
| $x$ | 1 | | | 1 |
| | $\overline{z}$ | $\overline{z}$ | $z$ | $z$ |

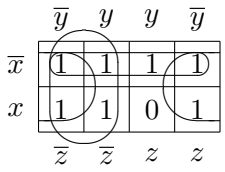
addend = $x\overline{y}$

**Exercise** Write out all the $2 \times 1$ and $2 \times 2$ patterns that can occur in a 3 variable Karnaugh map, and give the addends they generate.

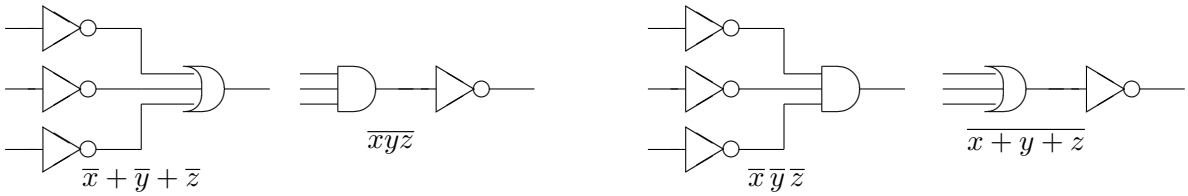
**Exercise** Draw a Karnaugh map and a corresponding minimized DNF for $x\overline{y} + xy\overline{z} + xyz$.

Note that a minimal DNF does not necessarily give the smallest logic circuit.

**Example** The Karnaugh map for $\overline{xyz}$ is




which gives the minimal DNF $\overline{x}+\overline{y}+\overline{z}$. The corresponding logic circuit, using multi-input gates, is given on the left below, while on its right is a smaller equivalent circuit which does not correspond to any DNF. A similar situation holds for $\overline{x+y+z}$ which has minimal DNF $\overline{x}\,\overline{y}\,\overline{z}$.




## 5.3.6 Extensions of the Karnaugh approach

Karnaugh maps for 4, 5 and 6 variables are possible, but as the number of variables increases they become less easy to write out and use. There is an algorithm, the Quine–McCluskey algorithm, which is used in practice for expressions with 4 or more variables. We will not discuss it in this course. However, you should be aware that, as we have seen, there are $2^n$ minterms so any algorithm which needs to consider them all is going to become impractical for relatively small values of $n$. Furthermore, the problem of find a minimal logic circuit for any given expression is known to be NP-complete, there is no known polynomial time solution. For this reason heuristic methods which give a reasonable optimal solution are often preferred.

We should also note that in circuit design the problem is not always specified by a completely defined logical expression. The specification may be of the form, say:

> The circuit has $n$ Boolean valued inputs, $x_1, \ldots, x_n$. If $x_1$ and $x_2$ are true then the circuit must output true, if all the inputs are false then the circuit must output false, ... etc.

In effect the truth table is described. The point is that there are often values for which the outcome is not specified, i.e. the specifier 'doesn't care'. This may be specified by using # in the truth table.

For example, a designer may specify a circuit with 2 inputs and the following corresponding outputs

| $x$ | $y$ | $r$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | # |

The designer 'does not care' what value the implemented circuit returns when exactly two of its inputs are true.

The Karnaugh map (and Quine–McCluskey) minimisation methods can exploit this by leaving minterms whose value is # out of the DNF, but allowing these minterms to be included in the matching steps. This allows optimisation against the choice values for of the # valued minterms. In any final circuit the unspecified minterms will all have a value of 0 or 1, the method ensures the values taken are those which give the smallest circuit.

## 5.4 Computer integer arithmetic

We complete this chapter by discussing the implementation of integer addition and subtraction using logic circuits.

### 5.4.1 Integer addition - a full adder

Recall that we defined binary addition positionly, as for decimal addition, starting with the rightmost digits and 'carrying forward' when the digits sum to 2.

$$\begin{array}{cccccc} & 1 & 1 & 0 & 1 \\ + & 0_1 & 1 & 1 & 0 \\ \hline 1 & 0 & 0 & 1 & 1 \end{array}$$

For each column,

- ◇ if the sum of both entries and the carry value is 0, then the sum entry is 0.
- ◇ if the sum of both entries and the carry value is 1, then the sum entry is 1.
- ◇ if the sum of both entries and the carry value is 2, then the sum entry is 0 and there is a carry value of 1.
- ◇ if the sum of both entries and the carry value is 3, then the sum entry is 1 and there is a carry value of 1.

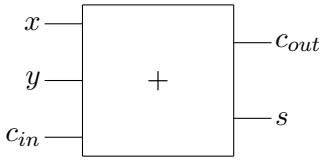
In fact we can see that each column sum involves the two entries and two carry values, the incoming carry value from the previous column sum, $c_{in}$, and the carry value to be passed to the next column, $c_{out}$. Denoting the two column values as $x$ and $y$ we have the following truth table which defines the sum digit, $s$, and $c_{out}$ for each of the possible values of $x$, $y$ and $c_{in}$.

| $x$ | $y$ | $c_{in}$ | $c_{out}$ | $s$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

The first step in constructing a logic unit to perform binary addition is to build a circuit which implements the expressions for $s$ and $c_{out}$. We can show that

$$s = xyc_{in} + \overline{x}\,\overline{y}c_{in} + \overline{x}y\overline{c_{in}} + x\overline{y}\,\overline{c_{in}}, \qquad c_{out} = xy + yc_{in} + c_{in}x.$$

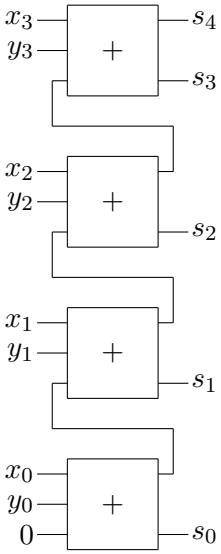
We can use these and the techniques defined earlier in this chapter to build the circuits. We write the result graphically as a box labelled + with three inputs on its left and two outputs on its right. This is called a full adder.



**Exercise** Use the techniques from earlier in this chapter to construct a logic circuit which computes $c_{out}$ and $s$ from the inputs $x$, $y$ and $c_{in}$.

We can connect circuits of this type together, one for each digit in the number, to get a *ripple carry adder*.

For example, for 4-bit integers $x_3x_2x_1x_0$ and $y_3y_2y_1y_0$, say, we have 8 inputs and the initial $c_{in}$ which is set to 0. There are 4 outputs $s_3$, $s_2$, $s_1$ and $s_0$ which are the bits of the sum, together with the final 'carry' or 'overflow' bit, which is the last value of $c_{out}$.




## 5.4.2 A half adder

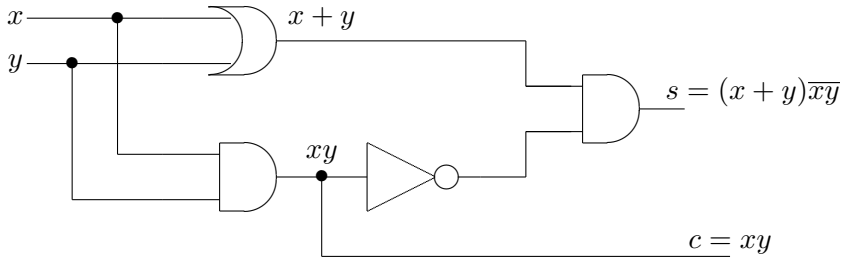
We can build a full adder from two half adders. A *half adder* just takes the two digits $x$ and $y$ as inputs, and calculates their sum as a result digit, $s$, and a carry bit, $c_{out}$. The truth table for the a half adder is

| $x$ | $y$ | $c$ | $s$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

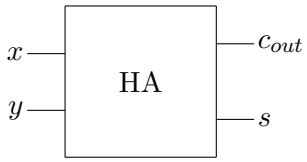
from which we get

$$\begin{aligned} c &= xy \\ s &= \overline{x}y + x\overline{y} \ . \end{aligned}$$

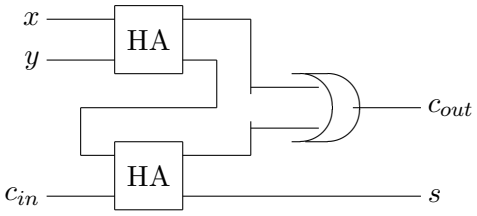
Noting that $\overline{x}y + x\overline{y} = (x + y)\overline{xy}$ we have the following logic circuit for the half adder



This is drawn as



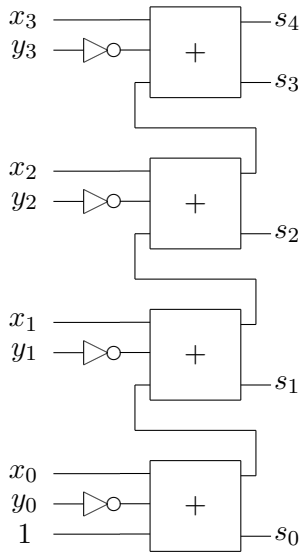

We can then use two half adders to build a full adder,




**Exercise**: Show that the truth table for this is the same as the initial truth table for full addition.

### 5.4.3 Subtraction

Recall that subtraction can be defined as adding a negation, and that in the two's complement notation the negation of a integer can be found by inverting the all its bits and then adding $o \ldots 01$. Thus we can implement a subtraction logic circuit using full adders and NOT gates, inverting the bits in $y$ and setting the first $c_{in}$ to 1. Below is the circuit for 4-bit integers.




**Exercises** For the following logical expressions, write down the Karnaugh map and use it to find a minimal corrsponding DNF.

1. $\overline{x}\ \overline{y} + \overline{x}\ y$
2. $\overline{x}\ \overline{y} + xy$
3. $x\overline{y}z + xyz$
4. $xy + xy\overline{z}$
5.

| $x$ | $y$ | $r$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

# Chapter 6

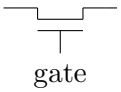
# Switching circuits

We now consider how logic gates are built from transistors. We will not look at the physics of transistors, the conductivity of metal, the behaviour of electrons etc. We shall use a Boolean model in which wires are thought of as connectable to 5V (1) and 0V (0) nodes (called rails), and switches (transistors) connect and disconnect wires creating potential conduction paths (current flow).

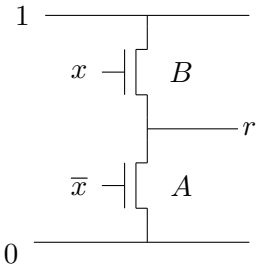
You can think of a transistor as a box with three connection points, terminals. One terminal is used to enable/disable current flow though the transistor. This is called the *gate*. If the transistor is enabled there is a *conduction path* through it. The other two terminals are called the *load* terminals. In a switching circuit we have a set of wires connecting each transistor terminal to another transistor terminal, to the 1 or 0 rail, or to nothing, i.e. 'floating'.

## 6.1 Simple circuits

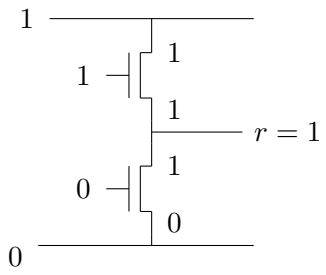
We shall start by looking at examples using transistors which are enabled when the value on their gate is 1. This type of transistor is shown graphically as



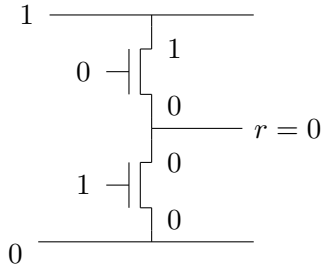

Consider the circuit below, in which the values on the gates are represented by the Boolean values $x$ and $\overline{x}$.



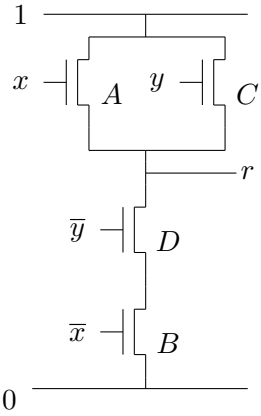

We can compute the value at $r$ for given values of $x$. The switch $A$ is connected to 0, so the value at its lower terminal is 0. Similarly the value at the higher terminal switch $B$ is 1. If $x = 1$ then $B$ is enabled and there is a conduction path through $B$, causing the value on its lower terminal to be 1. This is then also the value of $r$ and the upper terminal of $A$. The 0 on $A$'s gate disables $A$ so it is OK to have 0 on one load terminal and 1 on the other.




If $x = 0$ then $A$ is enabled and there is a conduction path through $A$, causing the value on its upper terminal to be 0. This is then also the value of $r$ and the lower terminal of $B$.

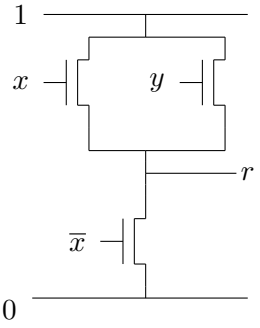



Now consider the circuit for which two input values $x$ and $y$ are needed to define the gate values.




The switches $D$ and $B$ are *in series* so there is a conduction path from the 0 rail to $r$ if they are both enabled. The switches $C$ and $A$ are *in parallel* so there is a conduction path from the 1 rail to $r$ if either of them is enabled.

It is necessary to design switching circuits with care. If a conduction path is created between 1 and 0 then the circuit will 'blow up'. For example, in the circuit



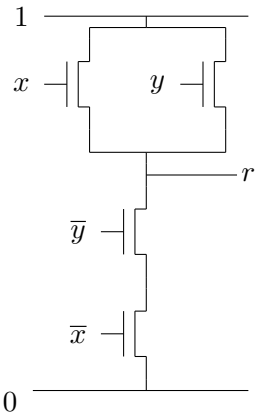

if we set $x = 0$ and $y = 1$ we have a conduction path from 1 to 0.

It is not always easy to spot potential 1-0 connections in a switching circuit. We shall consider a design methodology which addresses it later in this chapter.

### 6.1.1 Implementing logic gates

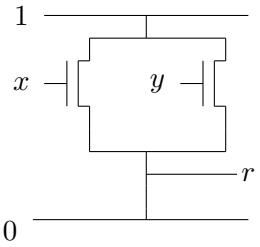
**an OR gate**

Consider again the switching circuit



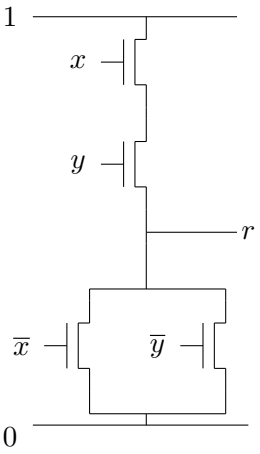

Notice that the value of $r$ is 0 if $x = y = 0$ and $r = 1$ otherwise. Thus the value on $r$ is $x \vee y$. So we can implement an OR gate as a switching circuit in this way.

Notice, we cannot simply implement an OR gate as

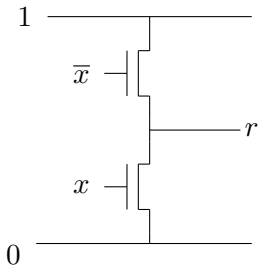



as when $x = 1$ there would be a conduction path between the 1 and 0 rails.

**an AND gate**



**a NOT gate**
If we swap the gate values of the first example above we get a NOT gate.




There is no reason why logic circuits must be built from logic gates. If we have a specific and heavily used logic expression we may be able to create a custom switching circuit which computes it more efficiently than one built from the standard logic gates. However, as we have said, in reality we do not want to allow a free choice of switching circuit because of the potential for connecting the 0 and 1 rails. Also, as a result of the transistor design technology, the type of switch we have considered so far may be less reliable at conducting 1 than 0. We now consider a second type of transistor and a circuit design methodology which makes the creation of a conduction path between 0 and 1 impossible.

## 6.2 N-type and P-type transistors

Formally, a *transistor* is a unit with three terminals, the gate and two load terminals.

The switches we have considered so far are *N-type* transistors, they are enabled if the value on their gate is 1.

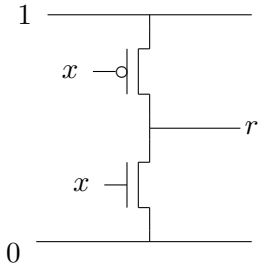
A *P-type* transistor is similar to an N-type except that it is enabled if the value on its gate is 0 and disabled if the value is 1.

We draw an N-type transistor as for the switches above, and a P-type transistor with an additional circle on the gate terminal to indicate that it is enabled on 0.


P-type


N-type

We can implement a NOT gate using N-type and P-type transistors as follows.



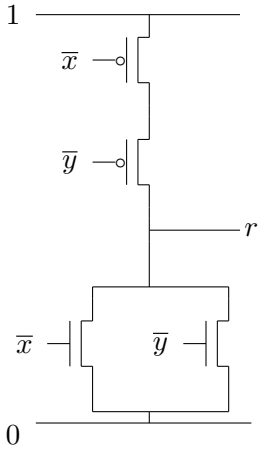

It is tempting to implement an AND gate in a similar way. However, real transistors are not perfect conductors. In detail,

- ◇ *N-type transistors*: enabled if and only if the value on the gate is 1, but are poor conductors of load 1
- ◇ *P-type transistors*: enabled if and only if the value on the gate is 0, but are poor conductors of load 0

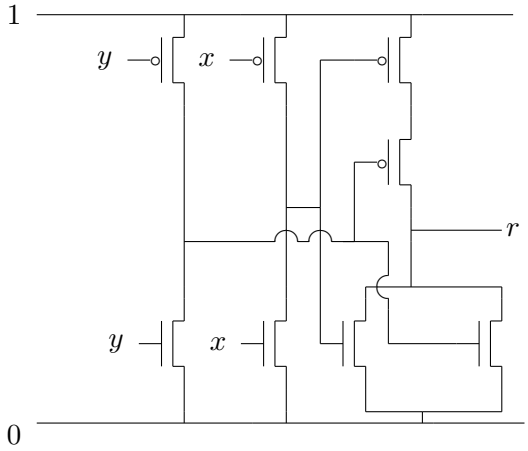
So, we design circuits so that all the transistors on conduction paths from 1 to the output are P-type and all transistors on conduction paths from 0 to the output are N-type. We say that such a network is *canonical*.

### 6.2.1 A canonical AND circuit

To obtain a canonical AND gate using N and P-type transistors, first we decide how to build it using both $x$,$y$ and $\overline{x}$,$\overline{y}$ as inputs. This can be done by modifying the simple switching circuit from above and using the identity $\neg\neg r = r$




We can convert this into a circuit with two inputs $x$ and $y$ by adding the corresponding NOT circuits and using the output of these circuits as the gate values.




**Exercise** Draw a canonical circuit which implements an OR gate.

## 6.3 Building canonical circuits

Recall that we want circuits in which all the transistors between the 1 rail and the output are P-type and all the transistors between the 0 rail and the output are N-type.

We achieve this by implementing the circuit for a logical expression $r$ in two parts. We design, using only P-type transistors, a circuit in which there is a conduction path through the circuit for exactly those input values for which $r = 1$. Initially we use both $x$ and $\overline{x}$ inputs if necessary. This is called the *pull-up* circuit.

We then design another circuit using only N-type transistors, a circuit in which there is a conduction path through the circuit for exactly those input values for which $r = 0$. This is called the *pull-down* circuit.

### 6.3.1 Constructing a pull-up circuit

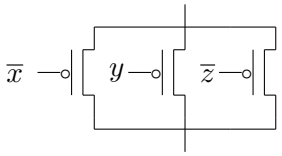
Given a logic expression composed of AND, OR and NOT gates we construct a pull-up circuit inductively on the structure of the expression.

- ◇ For an instance of a variable $x$ use a P-type transistor whose gate value is $\overline{x}$
- ◇ For an instance of a variable $\overline{x}$ use a P-type transistor whose gate value is $x$
- ◇ For a product $r_1 \dots r_k$ construct circuits for each $r_i$ and put them in series
- ◇ For a product $r_1 + \dots + r_k$ construct circuits for each $r_i$ and put them in parallel

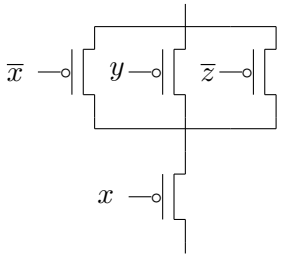
By construction there is a conduction path through the circuit for precisely those input values for which $r = 1$.

**Example** Construct a pull-up circuit for $r = (x + \overline{y} + z)\overline{x} + z$

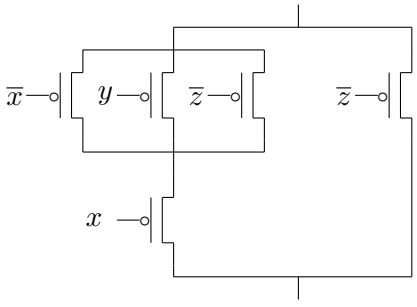
We have two expressions, $(x + \overline{y} + z)\overline{x}$ and $z$ to put in parallel. For $(x + \overline{y} + z)\overline{x}$ we need $x + \overline{y} + z$ which is




Next we put this in series with a transistor for $\overline{x}$.




Finally we put this in parallel with a transistor for $z$.




### 6.3.2 Constructing a pull-down circuit

Given a pull-up circuit we construct a corresponding pull-down circuit by inverting everything.

Replace each P-type transistor with an N-type transistor. Take the sequences of sub circuits which are in series and put them in parallel and take the sequences which are in parallel and put them in series.

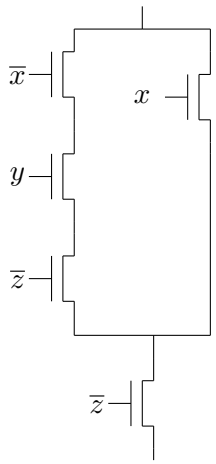
The result is a circuit which has a conduction path through it for precisely those input values for which $r = 0$.

You may find it easier to construct the pull-down circuit directly:

- ◇ For an instance of a variable $x$ use a N-type transistor whose gate value is $\overline{x}$
- ◇ For an instance of a variable $\overline{x}$ use a N-type transistor whose gate value is $x$
- ◇ For a product $r_1 \ldots r_k$ construct circuits for each $r_i$ and put them in parallel
- ◇ For a product $r_1 + \ldots + r_k$ construct circuits for each $r_i$ and put them in series

**Example** Construct a pull-down circuit for $r = (x + \overline{y} + z)\overline{x} + z$

Take the circuit from the previous example, swap the transistor types and interchange the series and parallel structures.




The pull-down circuit is the *dual* of the pull-up circuit.

### 6.3.3 Constructing a canonical circuit

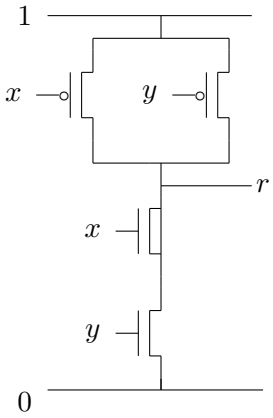
To construct a canonical circuit for an expression $r$ we take the pull-up circuit and attach the top wire to the 1 rail and the bottom wire to the output wire. For precisely those variable values for which $r = 1$ there will be a conduction path from 1 to $r$.

Then take the pull-down circuit and attach the bottom wire to the 0 rail and the top wire to the output wire. For precisely those variable values for which $r = 0$ there will be a conduction path from 0 to $r$.

Finally we can take the canonical NOT circuits for each negated variable, $\overline{x}$, and feed their output into the gates whose value is $\overline{x}$.

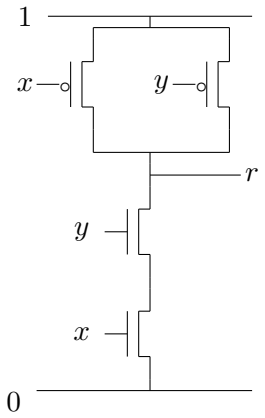
The circuit will be canonical by construction.

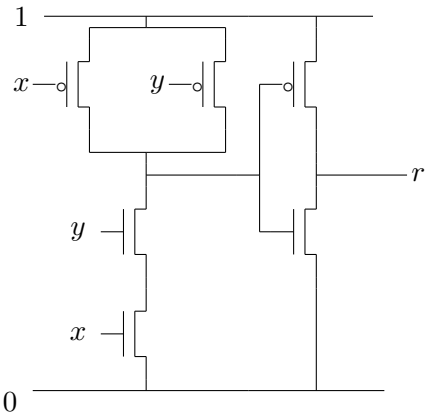
**Example** Consider $r = \neg(x \wedge y)$, a NAND gate. We can write this as $r = \overline{x} + \overline{y}$. Then we get the canonical circuit




**Exercises** Construct canonical circuits that implement:
1. $\overline{x}$ 2. $x + \overline{y}$ 3. $xz + yw$ 4. $x \Rightarrow y$ 5. $x\overline{y}$
6. $(\overline{x} + y)z$ 7. $x + \overline{y}z$ 8. a NOR gate

Note, if only a small proportion of the variables in a logical expression are negated then it may be more efficient to implement the negation of the whole expression and then negate the final result. For example, $\neg(x \wedge y) = (\neg x \vee \neg y)$ has canonical implementation on the left below and, since $x \wedge y = \neg(\neg x \vee \neg y)$ the circuit on the right is a corresponding AND gate.




## 6.4 Terminology

We conclude this chapter with some terminology which is used in various parts of the computer hardware industry.

The N-type and P-type transistors are MOSFET transistors, Metal-Oxide-Semiconductor Field-Effect Transistors.

P-type transistors are also called PMOS, or P-channel, or PFET transistors.

N-type transistors are also called NMOS, or N-channel, or NFET transistors.

The transistor circuits we have described are sometimes referred to as CMOS logic, Complementary Metal-Oxide-Semiconductor logic.

The phrase "metal-oxide-semiconductor" is a reference to the physical structure of certain field-effect transistors, having a metal gate electrode placed on top of an oxide insulator, which in turn is on top of a semiconductor material.

# Chapter 7

# Assembly language

Ultimately computers execute instructions which are coded as sequences of 0s and 1s. It is possible to write a program in this way, although extremely tedious and error prone. In order to make this task easier machines are provided with a mnemonic form of the instructions, and a program which reads a file written in this form and translates it into the corresponding binary representation.

This mnemonic form is the machine's *assembly language* and the program which translates the assembly code into binary is called the *assembler*.

It is also possible to write programs in the machine's assembly language, but generally we use more human oriented languages such as Java. Such languages are often called high level languages, and assembly languages are called low level languages. Compilers take a program written in a high level language and translate it into the machine's native assembly language.

High level languages typically have a rich structure of conditional statements, loops, data structures, and so on, which makes writing complex software systems a lot easier. Low level languages have only basic arithmetic, memory copy and branch instructions.

Although most programming these days is done in high level languages, assembly programming is still necessary. If you want to write a compiler, you need to understand assembly language (the compiler's target language) in order to write a program to generate it. Some high performance embedded system applications need to be coded at assembly level because of the limitations on code size or power consumption. Device drivers and other items very close to the hardware are also sometimes written in assembly code. Finally, if you want to understand CPU design, or simply tweak your code to optimise performance it, you need to understand how your CPU executes your programs.

In this chapter we give an introduction to assembly language programming using the MIPS language.

## 7.1 The MIPS processor

We cannot study every kind of processor since there is a huge variety. They appear not just in PCs but in all kinds of embedded systems (mobiles, TV, video recorders, washing machines, etc.). Nonetheless, like high level languages, studying one gives us some insight into all of them, so we shall focus on the MIPS processor family. It is a RISC (reduced instruction set) processor and has a small number of instructions which can be performed quickly and which make pipelining efficient.

We will use a simulator SPIM to simulate the MIPS processor and its assembly language.

If you want to run SPIM on your own computer you can download it from
`http://www.cs.wisc.edu/~larus/spim.html`.
There is a version of SPIM running on `teaching`.

### 7.1.1 Memory, bytes, and words

Recall from CS1801 the pigeonhole memory model. The computer's RAM is like a lot of pigeonholes, each of which can hold, in the case of MIPS, one byte, i.e. 8 bits, corresponding to numbers between 0 and 255. The pigeonholes are all arranged in order, and we number them from zero, so irrespective of the contents, each pigeonhole corresponds to a permanently allocated number called its address.

MIPS uses 32 bit addressing, i.e. the address is a 32 bit (4 byte) number. So MIPS can have up to $2^{32}$ addresses, namely 4096 Megabytes (MBytes) or 4 Gigabytes (Gbytes). A MIPS processor has a 32 bit architecture, i.e. data is handled in groups of 32 bits (4 bytes). A word for the MIPS processor is 4 bytes long.

When programming in machine code we need to specify memory addresses explicitly. In assembly language we can use labels, which are symbolic names for memory locations. The assembler works out the 32 bit address from this label. Before giving examples we need to discuss registers.

### 7.1.2 Registers

MIPS is a register based architecture. Registers are like memory locations but are positioned within the CPU so that access is fast. The MIPS processor has 32 registers, each storing a 32 bit word.

This value can be an unsigned integer, a (two's complement) integer, or an address, depending on the instruction.

Of these registers 24 are for general use. Their names are
`$v0, $v1, $a0 to $a3, $t0 to $t9, $s0` to `$s7`.
Arithmetic operations can only be performed on values in registers.

There are some conventions about register use, in particular `$a0, $a1, $a2, $a3` are used for arguments, parameters, of subroutines, `$v0, $v1` are used for results of expression evaluation, `$t0-$t9` are used for temporaries which are not preserved by subroutine calls. Understanding and using these conventions makes MIPS programs easier to read and maintain.

**Example**
```
li  $a0, 5
li  $a1, 6
add $a2, $a0, $a1
```
The instruction `li` is load immediate which puts the value on the right (5) into the register on the left (`$a0`). The instruction `add` adds, in this case, the contents of register `$a0` and register `$a1` and leaves the result in register `$a2`.

We don't go into instruction representation in this course but a brief discussion is needed to understand how the instructions are used. MIPS instructions (code) have to be represented as bit strings and it is efficient if they each fit in to one word, 32 bits. An instruction has an operator and values which are operated on, and if these values are themselves 32 bits long they cannot be included directly in the instruction. The 32 MIPS registers can each be specified using only 5 bits, and so the values are held in registers and only the register specifications need to be included in the instruction.

## 7.2 Data

Data and code are separated in a MIPS program. Variable declaration is the association of a name with a memory location so that the name can then be used in the program. Variable declarations should be introduced by the directive `.data`

```
    .data
X:  .word 5
```

This declares a variable `X` and initialises it with the value 5, represented as 32-bit string (`word`). In memory, 4 bytes is allocated and `X` is a synonym for the address of this location.

For arrays we want the array entries to be in adjacent memory locations. MIPS does not have an array type. Contiguous memory can be allocated using the `.space` directive.

```
    .data
A:  .space 40
```

This allocates 40 bytes, 10 words, of memory and `A` is a synonym for the address of the first of these words.

Values are copied into registers using the instruction `lw` (load word). This instruction takes a register `$r` and a memory address $x$ and copies the value held at the address $x$ into the register `$r`. The address $x$ is a 32-bit number and, as we mentioned above, representing the load instruction for the full memory range would take more than 32 bits. Thus the address $x$ is loaded in to a register `$s` and `lw` loads the contents of $x$ indirectly, loading the contents of the address held in `$s` (the contents of the contents of `$s`!) into `$r`. This is written

```
lw $r, ($s)
```

and read as 'load the contents of the address held in `$s` into `$r`'.

An address is put into a register using the instruction `la` (load address). (On the face of it `la`, and `li`, take 32-bit operands, however strictly speaking they are not pure MIPS instructions. They are built, by the assembler, using the primitive `lui` and `ori` instructions which load 16 bits and pad the rest. For this course you can treat `la` and `li` as MIPS instructions and you do not need to know about `lui` or `oir`.)

**Example** The following adds 5 and 6 and leaves the result in `$t2`.

```
    .data
X:  .word 5
Y:  .word 6
    .text
    la    $a0, X
    la    $a1, Y
    lw    $t0, ($a0)
    lw    $t1, ($a1)
    add   $t2, $t0, $t1
```

The directive `.text` switches the mode back to program code. The instruction `la $a0 X` loads the address associated with `X` into the register `$a0`. The instruction `lw $t0 ($a0)` loads the contents of the location held in `$a0` into the register `$t0`.

The companion instruction `sw $t0 ($a0)` (store word) stores the contents of the register `$t0` at the location held in `$a0`

**Example**

```
    la    $a0, X
    la    $a1, Y
    lw    $t0, ($a0)
    lw    $t1, ($a1)
    add   $t2, $t0, $t1
    la    $a2, Z
    sw    $t2, ($a2)
   .data
X: .word 5
Y: .word 6
Z: .word 0
```

MIPS also provides a `move` instruction which copies the contents of one register to another. This instruction is expanded by the assembler into an addition operation, the move is on the left below and the implementation as add immediate is on the right.

```
move $rt, $rs                    addi $rt, $rs, 0
```

Recall the `.space` directive which allocates several adjacent bytes of memory. The full forms of the `lw` and `sw` instructions allow addresses to be specified with an offset. For example,

```
lw $r, 8($s)
```

loads, into `$r`, the word (4 bytes) which begins 8 bytes on from the address held in `$s`.

This allows arrays to be implemented. For example, we can implement an array $A = [1, 2, 3, 4]$ using

```
    la    $a0, A
    li    $t0, 1
    sw    $t0, ($a0)
    li    $t0, 2
    sw    $t0, 4($a0)
    li    $t0, 3
    sw    $t0, 8($a0)
    li    $t0, 4
    sw    $t0, 12($a0)
   .data
A: .space 16
```

## 7.3 Arithmetic operations

We have already seen `add` which adds the contents of two registers and stores the result in a third. This instruction also tests for overflow. There is a corresponding instruction `addu` (add untrapped) which does the same thing but does not check for overflow.

In its pure form we can implement subtraction as the addition of the negation, but there is a MIPS instruction for subtraction

```
sub $d, $s, $t
```

which subtracts the contents of register `$t` from the contents of register `$s` and stores the result in register `$d`.

Sometimes we just want to add a constant to a value. The instruction `addi` (add immediate) does this.

```
addi $d, $s, C
```

adds the (signed) integer `C` to the contents of register `$s` and stores the result in register `$d`.

MIPS also provides a multiplication instruction `mult`, however this is not completely straightforward to use because of the likely size of the result.

```
mult $d, $s
```

`mult` is 64-bit operation which multiplies the two 32-bit contents of registers `$s` and `$t` and stores the 64-bit result in two special locations `HI` which holds the top 32 bits and `LO` which holds the bottom 32 bits. The programmer has to convert these values to whatever form they need.

There are several other forms of the arithmetic operations and bitwise logical operations provided in MIPS, but in the brief description in this course we will not discuss these.

## 7.4 Commenting

Commenting is always an important task to carry out when coding. However, as it is very easy to lose your train of thought when writing in assembly language, it is even more important to do so in assembly. Comments commence with a `#` and are terminated by end-of-line.

```
    li $a0, 10
#   This is a very important comment
    la $a1, X # So is this...
```

## 7.5 Control flow

Programs written in high level languages such as Java do not generally start at the beginning of the code, run, and then finish at the end. Execution moves from one place to another, loops, and takes different paths depending on the values of the input variables.

Control flow in assembly language is more primitive. Lines of code are labelled and control flow consists of instructions to jump to labels.

### 7.5.1 Unrestricted jump

The simplest type of command is just to jump to a label. In MIPS the instruction is `j label`. On its own it is not very useful. In the following example the update to `$a0` is ignored.

**Example**

```
      .data
  X:  .word 5
      .text
      la $a0, X
```

```
        j loadX
       li $a0, 5
loadX: lw $t0, ($a0)
       add $t1, $t0, $t0
```

It is easy to write nonterminating code.

**Example**

```
loop: la    $a0, X
      lw    $t0, ($a0)
      add   $t1, $t0, $t0
      j     loop
      .data
X:    .word 5
```

### 7.5.2 Branch instructions

Branch instructions are conditional jumps, essentially `if` statements. There is a test for inequality or equality and a jump is performed if the test succeeds.

It turns out that for implementing many of the standard control flow operations it is most efficient to use a branch on failure. The test for inequality is done by `bne`.

```
bne Register1, Register2, label
```

If the contents of `Register1` are not equal to the contents of `Register2` then program execution jumps to `label`. Otherwise the `bne` statement is effectively ignored and control flow continues with the statement immediately following.

```
      li $t0, 10
      la $a1, n1
      lw $t1, ($a1)
      move $a0, $t0

loop: addi $t0, $t0, -1
      add  $a0, $a0, $t0
      bne  $t0, $t1, loop
      .data
n1:   .word 1
```

The corresponding branch-if-equal instruction is `beq`

```
beq Register1, Register2, label
```

If the contents of `Register1` is equal to the contents of `Register2` then program execution jumps to `label`.

There are also `bge` and `ble` which branch if the left operand is greater than or equal to, and less than or equal to, the right operand respectively. There are abbreviations for comparison with zero,

`bgez Register1, label` is the same as `bge Register1, 0, label`

### 7.5.3 The goto statement

Many programming languages (e.g. Fortran and C++) have a some form of goto statement which corresponds to jump. However, as early as 1968, Dijkstra raised concerns about the unrestricted use of jumps in his seminal paper 'Go To Statement Considered Harmful'. Modern programming styles and languages do not allow unrestricted jumps. Instead more restricted control flow statements, such as `for`, `while-do`, `if-then-else` etc., are used. These are translated into equivalent jump-based code by the compiler.

We now show standard implementations of high level control flow statements in MIPS.

### 7.5.4 IF statements

IF statements can be translated directly into branch statements. For example, the Java statement on the left corresponds to the MIPS statement on the right.

```
Java                              MIPS

if (a0 == 5) {                    bne $a0, 5, cont
       statement1; }              # Code corresponding to statement1
statement2;                 cont: # Code corresponding to statement2
```

There is no instruction which directly corresponds to if-then-else, instead we use a branch and a jump.

```
Java                                MIPS

if (a0 == 20) {                     bne $a0, 20, eLabel
       statement1; }                # Code corresponding to statement1
else {                              j cont
       statement2; }          eLabel: # Code corresponding to statement2
statement3;                     cont: # Code corresponding to statement3
```

### 7.5.5 While loops

A `while` loop has a Boolean valued condition and a statement. If the condition evaluates to true then the statement is executed and the condition is evaluated again. This is repeated until the condition evaluates to false. The corresponding assembly code has a label at the start of the condition test and an unconditional jump to this at the end of the statement. There is also a label at the start of the statement which is jumped to if the condition is true and a label which is jumped to otherwise.

```
Java                                      MIPS
while (a0 > 3) { body; }            loop: bgt $a0, 3, stmt
statement;                                j cont
                                    stmt: # Code corresponding to body
                                          j loop
                                    cont: # Code corresponding to statement
```

In fact it is more efficient to reverse the outcome of the Boolean test and perform the jump to the next statement if the test evaluates to false. This saves code space.

```
Java:                                    MIPS
while (a0 > 3) { body; }           loop: ble $a0, 3, cont
statement;                               # Code corresponding to body
                                         j loop
                                   cont: # Code corresponding to statement
```

### 7.5.6 for loops

A `for` loop is a type of a while loop in which the number of iterations is specified explicitly.

```
Java                                                    MIPS
for (a0 = 0; a0 < 20; a0++) { body; }                   li $a0, 0
                                                  loop: bge $a0, 20, cont
                                                        # Code corresponding to body
                                                        addi $a0, $a0, 1
                                                        j loop
                                                  cont: # rest of code
```

The MIPS code is implementing the `while` loop equivalent of the `for` loop, which in Java has the form

```
a0 = 0;
while (a0 < 20) {
           body;
           a0++; }
```

## 7.6 The SPIM simulator

We have introduced instructions, such as `li`, `add`, etc. and we have introduced assembler directives, so we can store data in a particular memory location and use a label to refer to it. To actaully run code we would need a MIPS processor. In this course we look at the form of MIPS input which can be read and executed by a MIPS processor simulator, SPIM. Using SPIM we can step through an assembly language program as though it were actually running on a MIPS processor.

## 7.7 The structure of a SPIM input program

We need to specify the first instruction to be executed, and we need some print functions to allow us to output results and comments via SPIM. The default label of the start instruction is `main`.

A MIPS program can be input to the version of SPIM running on `teaching` in the following format.

```
        .text
main:
        (your code here)
        .data
        (your allocation of memory here)
```

### 7.7.1 System calls

SPIM translates the assembler into binary, the loading of the code into memory, the allocation of memory locations to variables and the actual execution of the code. The SPIM instruction `syscall` causes SPIM to access operating system services. The required service is specified by the value held in register `$v0`. There needs to be at least one `syscall` at the end of a SPIM input file which stops the program. In this course we shall need to read and print values and to stop the program execution.

As we have said, the behaviour of `syscall` depends on the current value of `$v0` which must therefore be set in the code. If the value in `$v0` is 10 then `syscall` stops the program.

**Example**

```
        .text

        la   $a0, X
        la   $a1, Y
        lw   $t0, ($a0)
        lw   $t1, ($a1)
        add  $t2, $t0, $t1
        la   $a2, Z
        sw   $t2, ($a2)
        li   $v0, 10
        syscall
       .data
   X: .word 5
   Y: .word 6
   Z: .word 0
```

You can type the above program into a file, `example1.a` say, and run SPIM.

```
teaching$ spim example1.a
```

To see some output from the program we use the parameter 1 which causes `syscall` to print out the value in register `$a0`.

```
        .text
main:
        la   $a1, X
        lw   $a0, ($a1)
        li   $v0, 1
        syscall       #print out the current value in $a0
        li   $v0, 10
        syscall       #halt the program
       .data
   X: .word 5
```

If you run SPIM on this file the number 5 will be output on the command line. There is no newline between the output and the command prompt. We can get SPIM to output a newline as a print string, as we shall discuss in more detail below. We declare a variable `cr` which holds the string `\n`.

```
        .text
main:
        la   $a1, X
```

```
        lw   $a0, ($a1)
        li   $v0, 1
        syscall       #print out the current value in $a0
        la   $a0, cr
        li   $v0, 4
        syscall       #print out the current value in $a0
        li   $v0, 10
        syscall       #halt the program
       .data
  X: .word 5
 cr: .asciiz "\n"
```

Be careful not to confuse an address and its contents. The following code will cause SPIM to print out the address `X` not its contents as maybe mistakenly expected.

```
        .text
main:
        la   $a0, X
        li   $v0, 1
        syscall       #print out the current value in $a0
        la   $a0, cr
        li   $v0, 4
        syscall       #print out the current value in $a0
        li   $v0, 10
        syscall       #halt the program
       .data
  X: .word 5
 cr: .asciiz "\n"
```

### 7.7.2 Outputting character strings

It is useful to have SPIM print out messages in string format rather than in the numeric representation. We use this feature for formatting and printing out comments. The directive `.asciiz` is used to load a value which represents a specified character string. The string is specified using double quotes, and SPIM loads the correct corresponding value. The parameter 4 to `syscall` causes the value to be converted back to a character string for printing.

If the following code is run with SPIM, `Hello World!` will be printed on the command line. This string is stored in the variable `str1` along with a following newline.

```
        .text
main:
        la $a0, str
        li $v0, 4
        syscall # print the string "Hello World!"
        li $v0, 10
        syscall #halt program
       .data
 str:  .asciiz "Hello World!\n"
       # .asciiz is a directive to indicate that the following
       # is to be treated like a character string (null-terminated)
```

Don't forget to be careful with loops, as running the following program in SPIM will demonstrate.

**Example**

```
            .text
    main:
    loop:  la $a0, string
           li $v0, 4
           syscall # Display String
           j loop
           li $v0, 10
           syscall # halt program
          .data
string: .asciiz "Hello again!\n"
```

**Exercise** Work out the behaviour of the following program and then run it through SPIM to check your conclusions.

```
       .text
main:
       li $t0, 0
       la $a0, X
       lw $t1, ($a0)
loop:  add $t0, $t0, $t1
       addi $t1, $t1, -1
       bne $t1, 0, loop
       move $a0, $t0
       li $v0, 1
       syscall # display $a0
       li $v0, 10
       syscall # halt program
       .data
    X: .word 4
```

### 7.7.3 Selected assembly instructions

The following is a table of the SPIM `syscall` operations we shall use in the examples in this course.

| $v0 | Task executed |
|---|---|
| 1 | Read value stored in $a0 and print it out. |
| 4 | Read address stored in $a0 and print out character string starting at that address until a zero byte is encountered. |
| 10 | Stop program. |

The following is a list of some MIPS instructions with descriptions of their behaviour.

```
 add d, s1, s2          d = s1 + s2
 sub d, s1, s2          d = s1 - s2
  li d, c               load immediate (move constant i to register d)
addi d, s1, i           d = s1+i
   j label              jump to the instruction at the label
```

```
 beq s1, s2, label      branch  conditionally if s1 = s2
beqz s, label           branch if s = 0
 bge s1, s2, label      branch if s1 >= s2
bgez s, label           branch if s >= 0
  la d, address         load address (load the address of the
                                 given location to register d)
  lb d, address         load byte
  lw d, address         load word
  sw d, address         store word
move d, s               move contents of register s to register d
```

There are similar definitions for `bgt` and `bgtz` ($>$, greater than), `ble` and `blez` ($\leq$, less than or equal to), `blt` and `bltz` ($<$, less than), `bne` and `bnez` ($\neq$, not equal to).

# Chapter 8

# Finite state automata and regular languages

We have seen that computers can be built from very simple structures, for example NAND gates. The power of a computer comes from the way the gates are combined and used. We can ask the opposite question, is there any limit to the things computers can do? This question has its origins in the early 20th century with Godel's Incompleteness Theorem. We shall return to this at the end of the course. To study the potential and limitations of computation it is instructive to consider a machine abstractly as having internal state, and computation as moving from one state to another in response to input. This notion is formalised as automata theory.

In this chapter we study the simplest form of automaton, a finite state automaton, NFA.

As well as abstract models of computation, finite state automata are used directly in compilers. They are used to specify and recognise the 'words' of the programming language. Most modern programming languages allow the user to have any string of letters and integers (beginning with an integer) as the name of a program variable. There are infinitely many such strings (although in practice on any given computer there will be a maximum length of string that can be stored). Since the string length is a computer-defined limitation, the programming language usually allows any such string to be a variable name.

In general a program is a sequence of *words*, some of these words look like English words, such as the keywords `while` and `function`, but often other character strings such as `<=` are words. To run a program on a computer we first have to get the computer to read in the program and translate it into the computer's execution language. This process is known as *compilation*.

When a compiler reads an input program it sees a stream of input characters, and one of the first things it has to do is to identify the words. The person who designed the programming language has to 'tell' the compiler what the words in the language are, and this means defining some sets.

One way to define a class of sets is to use a regular expression, and this is the method that language designers often use for defining the words of a programming language. In this chapter we shall discuss regular expressions, and the finite state automata which represent them.

(You can read more about regular expressions and finite state automata in the book by Hopcroft and Ullman listed at the end of the Introduction, although the notation and conventions used in that book are slightly different from those we use here. You can read more about computer language definition and recognition in Compiler Theory: Principles, Tools and Techniques, by Aho, Sethi and Ullman, although this book is more advanced than is required for this course.)

Finite state automata and regular expressions turn out to be different ways of looking at the same thing. We begin by discussing regular expressions.

## 8.1 Regular expressions

An **alphabet** is any set of symbols. For computer languages the alphabet is usually a set of things that you can type from a keyboard, such as the ASCII characters, but any finite set of symbols can be used.

Words on the alphabet are strings of elements (letters) from the alphabet. So if $A = \{a, b, c, d, 1, 2\}$ is our alphabet then

$$abba, \quad b, \quad 1ab2, \quad ccc$$

are all words on $A$.

**Definition** The set of all strings on an alphabet is called the *Kleene closure*, and it is written $A^*$. So we have

$$A^* = \{a_1 \dots a_n \mid n \in \mathbb{N}, a_i \in A\} \quad \text{and} \quad X^* = \{u_1 \dots u_n \mid n \in \mathbb{N}, u_i \in X\}, \quad \text{where } X \subseteq A$$

The string with no letters is a string, and it is written $\epsilon$.

**Definition** If we have two strings we can form another one by *concatenating* them, writing them next to each other. So we have that $abba1ab2$ is the concatenation of $abba$ and $1ba2$.

The empty string acts like 1 in multiplication, for any string $\alpha$, we have $\alpha\epsilon = \alpha = \epsilon\alpha$.

We use common abbreviations from the standard multiplication notation, $a^3$ for $aaa$ etc.

**Definition** For sets $X$ and $Y$ we write

$$XY = \{\alpha\beta \mid \alpha \in X, \beta \in Y\}$$

So $XY$ is the set that can be obtained by concatenating an element from $X$ with an element from $Y$.

For example,

$$X = \{abba,\ ac1e,\ \epsilon\} \quad \text{and} \quad Y = \{cd,\ b,\ abba\}$$

$$XY = \{abbacd,\ ac1ecd,\ cd,\ abbab,\ ac1eb,\ b,\ abbaabba,\ ac1eabba,\ abba\}$$

Sometimes we want all the strings on an alphabet except for the empty string, and we have a special notation, $A^+$, for this:

$$A^+ = A^* \backslash \{\epsilon\}$$

For any set of strings $X$, $X^+$ denotes the set of all concatentations of one or more elements from $X$. If $\epsilon \notin X$ then

$$X^+ = X^* \backslash \{\epsilon\}$$

We can use regular expressions to denote certain types of subsets of the set of strings on a language $A$.

For the letters in $A$ we use the letter itself to denote the set which contains just that letter. So we write $a$ for the set $\{a\}$ which just contains $a$. Similarly, we write $\epsilon$ for the set $\{\epsilon\}$ which just contains the empty string.
(Careful, the set $\{\epsilon\}$ is NOT the same as the empty set, $\{\epsilon\}$ contains one element, $\epsilon$, but the empty set contains no elements. Just as $\{0\}$ is not the same as the empty set.)

We define other regular expressions recursively, so if we already have two regular expressions we describe how to construct others from these, using concatenation, set union and Kleene closure.

### 8.1.1 Formal definition of regular expressions

Formally, a regular expression denotes a set of strings and we define what a regular expression is by defining which set of strings it represents.

1. $\epsilon$ is a regular expression which denotes the set containing just $\epsilon$.

2. For $a \in A$, $a$ is a regular expression which denotes the set $\{a\}$.

   If $r$ and $s$ are regular expressions then:

3. $rs$ is a regular expression which denotes the set of strings which are formed by concatenating a string from $r$ with a string from $s$.

4. $r^*$ is regular expression which denotes the set of strings which are sequences of zero or more strings from $r$ concatenated together.

5. $r|s$ is a regular expression which denotes the union of the sets $r$ and $s$, the set of strings which are either a string from $r$ or a string from $s$.

By convention $*$ has highest priority then concatenation and then $|$. So $ab \mid aab^*$ means $(ab) \mid ((aa)(b^*))$.

### 8.1.2 Examples

The set of positive integers is

$$(1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^*$$

The set of integers is

$$(\ - \mid \epsilon\ )(1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^* \mid 0$$

The set of one or more $a$s followed by an arbitrary number of $b$s is

$$a\ a^*\ b^*$$

What is the set of elements denoted by $a(a^*|b^*)$

### 8.1.3 Equality of regular expressions

There may be many ways of denoting the same regular expression.

$a|a$ and $a$ both denote $\{a\}$, and $a(b|c)$ and $ab|ac$ both denote $\{ab, ac\}$.

**Definition** Two regular expressions, $r$ and $s$, are **equal** if they denote the same language.

$r|s = s|r, \quad r|(s|t) = (r|s)|t, \quad r(s|t) = (rs)|(rt), \quad r\epsilon = r, \quad$ etc.

We can write $r|s|t$ because $(r|s)|t = r|(s|t)$. To decide what $rs|t$ means we have operator priority, concatenation has higher priority than $|$. So $rs|t$ means $(rs)|t$ not $r(s|t)$.

**Exercises** Which of the following equalities are true? (i) $ab(ab)^* = (ab)^*ab$,
(ii) $ab(ab)^* = a(ab)^*b$, (iii) $ab(b \mid \epsilon)c = abbc \mid abc$, (iv) $a^*a^* = a^*$, (v) $aa^* = a^*$

## 8.2 Finite state automata

We can define sets using regular expressions, but if we use regular expressions to define things like the identifier names in a programming language then the compiler must be able to test whether a given particular input string belongs to a regular expression. For example, it needs to be able to tell that `fred` is an identifier name and `3451` is an integer.

It is not always easy even for a human to tell whether a string belongs to a regular expression. For example, do the strings `aaaabababa` and `aababababab` belong to the regular expression

$$a^* \ (\ (a\ b\ a)^* \mid b)\ (b\ a)^*$$

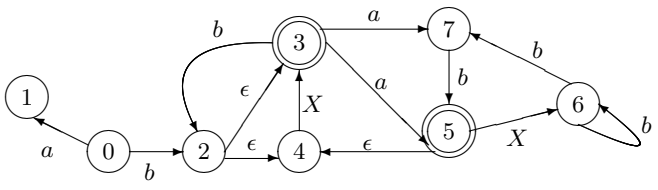
It can also be hard to describe all the strings of a regular expression. For example, what strings are denoted by

$$(\ 0\ 0^* \mid 1\ 0 \mid 1\ 1\ (0\ 0 \mid 1)^*)(1^* \mid 0\ (0 \mid 1\ 0)^* \mid 0\ 1\ 1)^* \mid 1\ 0^*$$

It turns out that regular expressions correspond to finite state automata, and we can use finite state automata to test whether or not a given string belongs to a given regular expression.

A finite state automaton (FA) is a directed graph whose edges are labelled with alphabet letters or $\epsilon$. The edges are often call transitions and the nodes are often called states. One of the nodes is the *start* state and some of the nodes are *accepting* states.

The following is a finite state automaton. The start state is state 0 and the accepting states, 3 and 5, are denoted by double circles.




Then $\delta(0, a) = \{1\}$, $\delta(2, \epsilon) = \{3, 4\}$, $\delta(2, a) = \emptyset$, etc.

**Definition** Formally, a finite state automaton (FA) is a set, $V$, of states, a set $A$ of symbols (letters), a transition relation $\delta \subseteq (V \times (A \cup \{\epsilon\} \times V)$ (the edges with their labels), a start state $s_0 \in V$ and a set $F \subseteq V$ of accepting states. An FA is often written as a 5-tuple $(V, A, \delta, s_0, F)$. This formal definition is needed for reason about FAs.

### 8.2.1 Traversing an FA

We can *traverse* an FA using an input string, that is a sequence of letters terminated by the special end-of-string symbol \$.

For example, we traverse the above FA with the string `babXbbbX$` as follows. We start in the start state, 0, and read the first symbol $b$. There is a transition labelled $b$ from state 0 to state 2, so we go to state 2 and read the next input symbol, $a$. There are $\epsilon$ transitions from state 2 to states 3 and 4, we choose to go to state 3. We then choose to follow the transition labelled $a$ to state 7, and we read the next input symbol $b$. We then go to state 5 and read the next input symbol $X$, then we go to state 6 and read the next input symbol, $b$. We choose to go round the loop back to state 6, and we read the next input symbol $b$. We then choose to go to state 7 and read the next $b$, then we go to state 5 and read the next input symbol, $X$. We could choose to go to state 6 and read the next input symbol, but we don't. We choose to follow the $\epsilon$ transition to state 4. Because it is an $\epsilon$ transition we don't read the next input symbol and the current symbol is still $X$. We now go to state 3 along the $X$ transition and read the last input symbol, \$. State 3 is an accepting state and the current symbol is \$, so the string is accepted and the process reports success.

Formally:
• At any given point in the FA traversal process there is a current state, $s$, and a current input symbol, $a$.
• At the start of the process $s$ is set to be the start state and the current symbol, $a$, is set to be the first symbol of the input string.
• At each step in the process we can choose to change the current state to be any state which can be reached from the current state along a transition labelled with the current input symbol or with $\epsilon$.
• If we move along a transition labelled with the current input symbol then the next input symbol is read and set to be the current input symbol.
• If the current state is an accepting state and the current input symbol is the end-of-string symbol \$, the input is accepted and the process terminates and reports success.
• If there are no transitions from the current state labelled with the current input symbol or $\epsilon$ then the process terminates and the input string is not accepted, unless the current input symbol is \$ *and* the state is an accepting state.

For example, if we traverse the above FA with the input string `a$` we end up in state 1 with the current input symbol \$. But since 1 is not an accepting state, the string is not accepted.

We now have another way of defining a set of strings, we can use FAs.

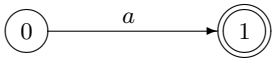
**Definition** The set of strings which are accepted by an FA is the *language* defined by that FA.

**Exercise** Traverse the above FA with inputs: $a\$ \quad b\$ \quad ba\$ \quad bba\$ \quad babXbbbX\$$

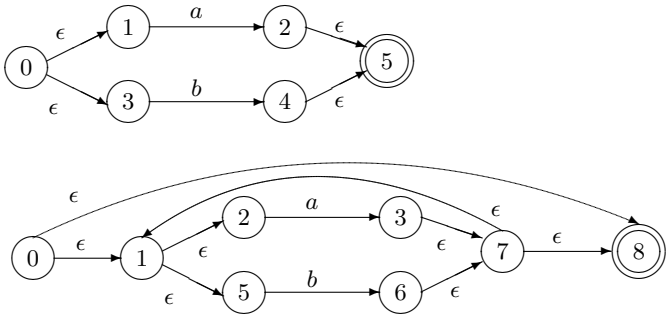
## 8.3 Thompson's construction

In this section we shall show that for every regular expression $r$ there is an FA which has the set denoted by $r$ as its language. Furthermore, we shall give an algorithm for constructing such an FA.

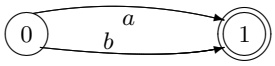
It is easy to see that the following DFA, whose start state is 0, has language $\{a\}$, the set denoted by the regular expression $a$.




It is also easy to see that the following FAs have languages $(a \mid b)$ and $(a \mid b)^*$ respectively.




These aren't the most obvious FAs. If we just wanted the FA for $(a \mid b)$ we would probably write

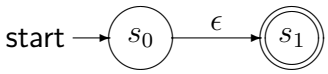



But the point is that the above FAs can be constructed by a formula that applies to any regular expression. (If $r$ and $s$ are regular expressions for which we already have FAs then we can generate an FA for the regular expression $(r \mid s)$ by making new start and accepting states and join these to the start and accepting states of the FAs for $r$ and $s$, see below). If there is a formula for the construction then this can form the basis of a computer program which *automatically* constructs an FA for a regular expression.

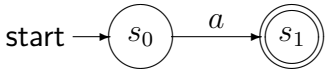
Once the computer can construct the FA we can also write a program to traverse the FA with an input string and we will have a program the can recognise the words of a programming language!

The formula for constructing an FA for a regular expression is inductive and uses the inductive definition of regular expressions. The method is call Thompson's Construction.

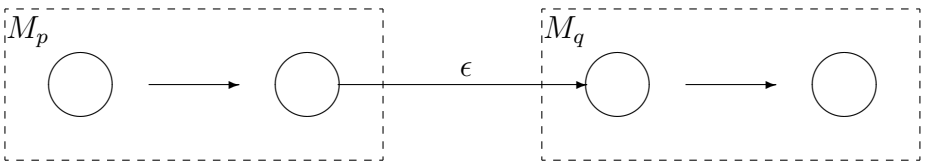
The regular expression $\epsilon$ corresponds to the transition diagram




The regular expression $a$, where $a \in A$, corresponds to the transition diagram

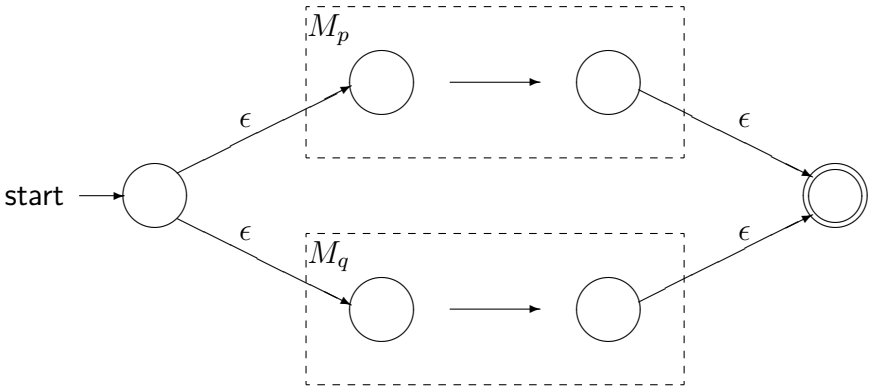



If $p, q$ are regular expressions with corresponding NFAs $M_p, M_q$ then:
$pq$ (concatenation) is represented by



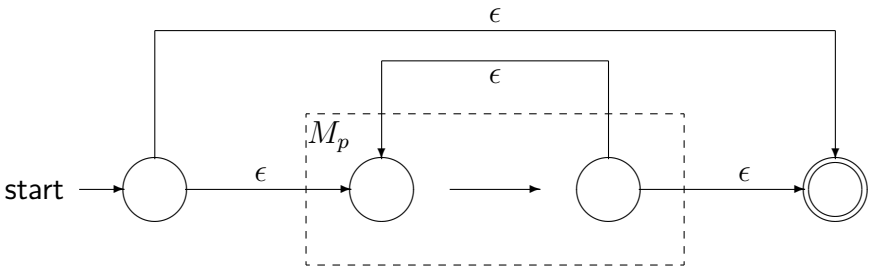

that is the two machines in series joined by an empty transition.

$p \mid q$ (alternation or union) is represented by



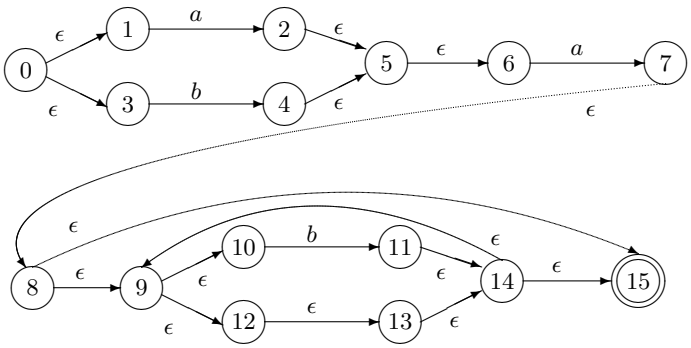

the two machines in parallel joined by empty transitions to a new start state and a new final state.

$p^*$ (Kleene closure) is represented by




the machine for $p$ with new start and finish states, all joined by empty arrows.

**Example** Using Thompson's construction for the regular expression $(a \mid b)\ a\ (b \mid \epsilon)^*$ we get the FA




**Exercises** Use Thompson's construction to find an FA for the regular expressions: 1. $a \mid b$ 2. $(a \mid b)^*$ 3. $a \mid b \mid c$ (i.e. $(a \mid b) \mid c$) 4. $(a \mid b \mid c)^*$ 5. $(a \mid b)^* a$ 6. $a^* b^*$

## 8.4 Deterministic finite state automata (DFA)

We need to be careful with the above definition of the language of an FA. If a string is accepted by an FA traversal then it is in the language of the FA. However, if the string is not accepted by the FA it may still be in the language of the FA because there may be a different traversal which does result in acceptance.

The problem is that there are choices that can be made during a traversal. If a state has more than one transition labelled with a particular symbol then the traverser can

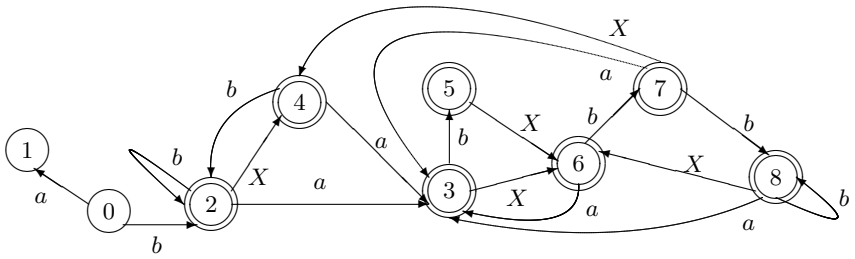
choose to take any of these transitions. Also, if the state has an $\epsilon$ transition then the traverser can choose this transition whatever the current input symbol is.

To be sure that when the traverser does not accept a string that string is not in the language we need the FA to be deterministic, i.e. for there to be no choices during the traversal.

**Definition** A *deterministic finite state automaton* is an FA with the property that there are no $\epsilon$ transitions and that for any symbol $a$ there is at most one transition labelled $a$ from each state in the FA.

Equivalently, the relation $\delta \subseteq (V \times (A \cup \{\epsilon\}) \times V)$ is actually a function $\delta : (V \times A) \to V$.

The above FA is not deterministic, but the following is a DFA which defines the same language.




## 8.4.1 The subset construction

The problem with FAs built by Thompson's construction is that they can be non-deterministic, and so if we have an input string which is not accepted by the FA it may none-the-less be a string in the language of the FA.

Deterministic FAs do not have this problem, because there is only one possible traversal of a DFA with a given input string. Every DFA is an FA, but there are FAs which are not DFAs. So it is possible, in principle, that there are languages which can be specified by an FA which cannot be specified with a DFA. However, it turns out that this is not the case, and that for every FA there is an equivalent DFA. Furthermore, there is an algorithm which, given an FA, constructs an equivalent DFA. This means that, for any regular expression, we can use a DFA to decide whether any given string is in the regular expression.

The algorithm which takes an FA and returns an equivalent DFA is called the subset construction. It works by combining together the FA states which can be reached on a given input.

Let $N$ be an NFA, and let $D$ denote the DFA we are trying to construct. The states of $D$ are sets of states of $N$. The sets are generated by looking for the subset of states that can be reached by making any number (including zero) $\epsilon$ transitions from states in $N$.

We need the following notation. If $s$ is a state in our NFA, $T$ is a set of such states, and $a \in A$ then

$\epsilon - closure(T)$ is the set of NFA states reachable from the NFA states in $T$ *via* $\epsilon$-transitions alone.

$Ta$ is the set of NFA states to which there is a transition on input symbol $a$ from some state $s$ in $T$.

The general algorithm for constructing the required DFA $D$ is as follows:

• The start state, $T_0$, of $D$ is $\epsilon$-closure($\{s_0\}$), the set of states which can be reached from the start of $N$ by just using $\epsilon$ arrows.

• Then for each $a \in A$, calculate $T_0a$ – the set of states from $N$ which can be reached from states in $T_0$ along an arrow labelled $a$.

• If $T_0a$ is not empty, form the $\epsilon$-closure of this by adding all the other states which can be reached from states in $T_0a$ along $\epsilon$ arrows. So we have a new state $T_1 = \epsilon$-closure($T_0a$) in $D$.

• We then put a transition labelled $a$ from $T_0$ to $T_1$, defining $\delta_D(T_0, a) = T_1$, and 'mark' the state $T_0$ as having been dealt with.

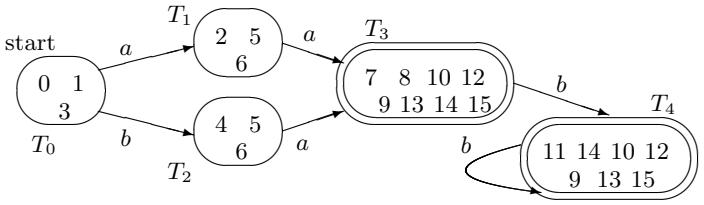
• We then repeat the process for each one the new states that we have constructed. So for every $a \in A$ we form $T_1a$, the set of states reachable using an $a$ arrow from a state in $T_1$, and then we calculate $\epsilon$-closure($T_1a$) etc.

• When all the states $T_i$ that we have constructed are also marked as having been dealt with then the construction of $D$ is complete.

• The start state of $D$ is the state $T_0$ which contains the start state of $N$. A state in $D$ is an accepting state if it contains at least one accepting state from $N$.

**Example** The subset construction on the NFA for $(a \mid b)\ a\ (b \mid \epsilon)^*$ given above gives the following sets.

$$
\begin{array}{rll}
T_0 = \epsilon\text{-closure}(\{0\}) = \{0,1,3\} & T_0a = \{2\} & T_0b = \{3\} \\
T_1 = \epsilon\text{-closure}(T_0a) = \{2,5,6\} & T_1a = \{7\} & T_1b = \emptyset \\
T_2 = \epsilon\text{-closure}(T_0b) = \{4,5,6\} & T_2a = \{7\} & T_2b = \emptyset \\
T_3 = \epsilon\text{-closure}(T_1a) = \{7,8,9,10,12,13,14,15\} & T_3a = \emptyset & T_3b = \{11\} \\
\quad \epsilon\text{-closure}(T_2a) = T_3 & & \\
T_4 = \epsilon\text{-closure}(T_3b) = \{11,14,15,9,10,12,13\} & T_4a = \emptyset & T_4b = \{11\} \\
\quad \epsilon\text{-closure}(T_4b) = T_4 & &
\end{array}
$$




**Exercises** Use Thompson's construction and the subset construction to find a DFA for each of: 1. $a \mid b$ 2. $(a \mid b)^*$ 3. $(a \mid b \mid c)^*$ 4. $a^*b^*$

## 8.5 A lexical analyser

We can now see how we could write a computer program which takes as input a set defined using a regular expression and a string, and output 'accept' if the string is in the set denoted by the regular expression and output 'reject' otherwise.

We use Thompson's construction to construct an FA and then the subset construction to construct an equivalent DFA.

We then write the DFA as a table. The rows of the table are indexed by the states of the DFA, by convention the first row is indexed by the start state, and the columns of the table are indexed by the alphabet symbols and the special end-of-file symbol \$.

It there is a transition labelled $a$ from state $s$ to state $t$ then we put $t$ in row $s$, column $a$ of the table. If $s$ is an accepting state then we put $acc$ in row $s$, column \$.

So the table version of the DFA from Section 8.4 is

| | \$ | $a$ | $b$ | $X$ |
|---|---|---|---|---|
| $T_0$ | | $T_1$ | $T_2$ | |
| $T_1$ | | | | |
| $T_2$ | acc | $T_3$ | $T_2$ | $T_4$ |
| $T_3$ | acc | | $T_5$ | $T_6$ |
| $T_4$ | acc | $T_3$ | $T_2$ | |
| $T_5$ | acc | | | $T_6$ |
| $T_6$ | acc | $T_3$ | $T_7$ | |
| $T_7$ | acc | $T_3$ | $T_8$ | $T_4$ |
| $T_8$ | acc | $T_3$ | $T_8$ | $T_6$ |

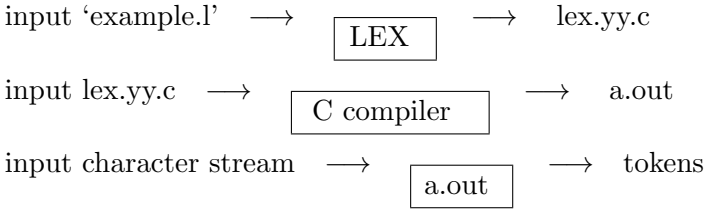
Then a simple program walks the table using an input string.

```
Input: a DFA, with start state s0, written as a table T, and a string
a1 a2 ... an $

Set symbol := a1;  state := s0; flag := 1

While (flag = 1) {
     if (T(state, symbol) is empty) { set flag := 0}
     else  { if (symbol = $) {set flag := 0} }
             else { state := T(state,symbol)
                    symbol := next input symbol; }
           }

if (T(state,current) = acc) then {return accept}
else {return reject}
```

## 8.6 The LEX lexical analyser generator

The UNIX system has a program LEX which does exactly what we have described above. You give it a regular expression and it constructs a DFA based recogniser.

Using the editor, create a file 'example.l' that is the input to LEX. Then type 'lex example.l' and LEX creates the file lex.yy.c which is written in C. This file is then run through a C compiler, you can use the command 'gcc lex.yy.c -ll', which produces a program called a.out. This is the lexical analyser.




An input file for LEX, a LEX program, has three parts: declarations, rules and auxiliary procedures. Each part of the program is separated by a line '%%'.

LEX program format:

declarations
%%
rules
%%
auxiliary procedures

Any of the three parts may be empty but the first line of separators cannot be left out, and we shall not be using the auxiliary procedures section.

The declarations section containd the definitions of the regular expressions. The rules section is a list of statements of the form $p$ $\{action\}$, where *action* is a program fragment written in C that describes what should be done if an input string belongs to $p$. The third section contains any auxiliary procedures that may be needed by the actions.

LEX is designed to generate the first stage of a compiler and the rules and procedures sections support this role. In this course we simply use LEX to experiment with regular expressions and we only use the rules section, and the productions section to print out a message.

You can give LEX several regular expressions and it will test each in turn to see if it can find one that matches your input string. The following is a very simple LEX program that generates a lexical analyser which recognises decimal numbers and identifiers composed of lower case letters and digits, beginning with a letter.

```
identifier [a-z]([a-z]|[0-9])*
number     [1-9][0-9]*(\.[0-9][0-9]*)?
%%
{number}       {printf("number\n") ;}
{identifier}   {printf("identifier\n") ;}
%%
```

Here we type `\.` because 'dot' is a special symbol in LEX and we want the literal. LEX has some shorthands, for example `[a-z]` stands for all of the lower case letters. LEX also uses ? for optional, so $(r|\epsilon)$ is written `r?`. Finally, so that LEX can decide when strings are the names of regular expressions these are enclosed in braces, `{}`, when they are used.

Create a file ex1.l, then type lex ex1.l. This produces lex.yy.c, so type gcc lex.yy.c -ll to produce a.out. Then if you type a.out and then 1.02 you will get 'number' printed on the screen. If you type 1.02 $fred1$ $temp1$ then 'number' 'identifier' 'identifier' will appear on the screen.

```
teaching$ lex ex1.l
teaching$ gcc lex.yy.c -ll
teaching$ a.out
teaching$ 1.02  fred1  temp1 = eas
number  identifier  identifier = identifier
```

Unrecognised character strings are echoed. So if you type < then < will be repeated back to you.

Try playing with LEX

## 8.7 Not every set can be defined by a regular expression

Unfortunately there are some sets that we need to study and use which cannot be defined using regular expressions, and hence which cannot be recognised using finite state automata.

For example, most programming languages allow you to write mathematical expressions such as

$$1 + fred - 3 * john$$

It is possible to write a regular expression which describes the set of all mathematical expressions which can be formed from integers, identifiers and $+, -, *, \div$. However, we need parentheses in the arithmetic expressions to over-ride operator priorities. For example, $3 * 4 + 1$ evaluates to 13, we need parentheses to specify $3 * (4 + 1)$ which equals 15.

It turns out that it is not possible to write a regular expression which specifies arithmetic expressions that include parentheses in the normal way. The parentheses must be balanced in the sense that every left parenthesis must have a matching right parenthesis, and this means that there must be the same number of left and right parentheses.

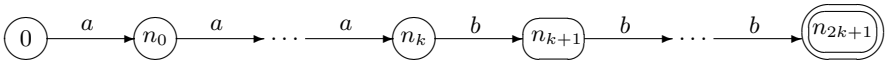
This is a general problem for regular expressions. For example, $a^*b^*$ is a regular expression whose set is the set of all strings $a^n b^m$ where $n, m$ are natural numbers. But there is no regular expression for the set of strings $a^n b^n$, which is any number of $a$s followed by the same number of $b$s.

We now give a proof of this fact, which is often expressed by saying 'regular expressions can't count'. This shows that not all sets can be defined using regular expressions.

**Theorem 1** *The set $\{a^n b^n \mid n \in \mathbb{N}\} = L$ is not the language of any DFA and hence cannot be defined using a regular expression.*

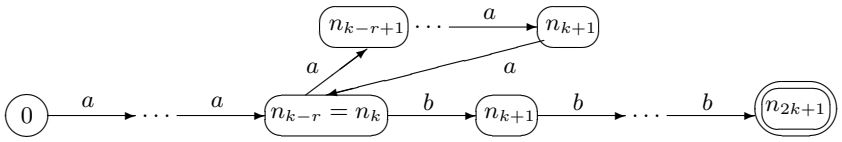
*Proof* The proof is by contradiction. We assume that there is a DFA which accepts precisely the strings of the form $a^n b^n$ and show that the DFA will also accept a string of the form $a^{n+m} b^n$ for some integer $m > 0$. This will then be a contradiction of the assumption that the DFA accepts only strings with the same number of $a$s and $b$s.

Suppose that there is a DFA whose language is $L$, and suppose that the DFA has $k$ states. The DFA must accept the string $a^{k+1} b^{k+1}$ so there must be a path through the DFA from the start state to an accepting state whose first $k+1$ edges are labelled with $a$ and whose second $k+1$ edges are labelled with $b$.




Since the DFA has only $k$ states, two of the states $0, n_0, \ldots, n_k$ must be the same. Suppose that $n_j = n_i$ where $j < i$. So $j + r = i$ and $r > 0$. Since there is only one edge labelled $a$ whose source state is $n_j$ we must have $n_{j+1} = n_{i+1}$. The same argument then gives that $n_{j+2} = n_{i+2}$. Carrying on in this way we see that we must have $n_{k-r} = n_k$.

Thus there is a path




in the DFA, and so the DFA also accepts the string $a^{k+1+r} b^{k+1}$. Since $r > 0$ this string is not in $L$, contradicting the assumption that the DFA accepts precisely the strings in $L$. Thus there can be no DFA whose language is $L$.

To specify the language of arithmetic expressions which include parentheses, we can use context free grammars, the topic of the next chapter.

## 8.8 Look up on the WEB

**Stephen Kleene** Born in America in 1909, Kleene proved that regular expressions and finite state automata are equivalent. It is also said that he discovered a previously unknown type of butterfly, which was named after him.

## 8.9 Exercises

1. Write out the set defined by the regular expressions
(i) $(a \mid b)(b \mid \epsilon)cb$, (ii) $(0 \mid 1)10(1 \mid 0)0$, (iii) $(a \mid a)ab(b \mid b)a$, (iv) $(abc \mid a)(b \mid \epsilon)bc$.

2.(i) Write down a regular expression for the set of all strings of $a$ of even length.
(ii) Write down a regular expression for the set of all strings of $a$ and $b$ that contain exactly one $a$.
(iii) Write down a regular expression for the set of all strings of $a$ and $b$ that start and end with the same symbol.

3. Write out the set defined by the regular expression $a \;\; b\ (b \mid \epsilon \mid a)\ (\ b \mid b\ )$.

4. Write down a regular expression whose set is the set of strings of 0s and 1s of even length.

5. Write down a regular expression whose set is the set of strings of 0s and 1s of odd length.

6. Write down a regular expression whose set is the set of strings which contain an even number of 0s and an even number of 1s.

7. Use Thompson's construction to construct an FA for the regular expression
$(\ a^*\ b\ (\ a \mid b\ )^*\ )\ (\ a \mid \epsilon\ ) \mid a$.

8. Use the subset construction to construct a DFA for the FA you constructed in Question 7.

9. Prove that the set $\{a^n\ b\ c^{2n} \mid n \in \mathbb{N}\ \}$ cannot be defined by a regular expression.

# Chapter 9

# Pushdown automata and Turing machines

In Chapter 8 we saw that regular expressions can be represented using finite state automata, and that these automata can be used to test whether a string belongs to a particular regular expression. We would like a similar method which allows us to recognise wider lasses of sets, including arithmetic expressions with brackets.

It turns out that bracket matching this is harder. For regular expressions we could construct a deterministic DFA which can then be used without backtracking in linear time. We can use push down automata, that is DFAs with a stack, to recognise languages with bracket matching, but PDAs are not always deterministic.

In this chapter we define and discuss PDAs, and then we look at Turing machines, which can describe all computable sets. You can read more about push down automata the book by Hopcroft and Ullman listed at the end of the Introduction, although again the notation and conventions used in that book are slightly different from those we use here.

## 9.1 Pushdown automata

A pushdown automaton (PDA) is a finite state automaton together with a stack. The actions performed are traversing transitions, as for FAs, and additionally pushing symbols on to the stack and popping symbols off the stack. The action taken depends on the symbol currently on the top of the stack, the current input symbol, and the current state of the FA.

For FAs we described the actions using the graphical representation of the FA. We could do this because the only actions are transitions, represented by arrows in the graph, and acceptance, represented with double lines on the nodes. For a PDA we also have to define stack push and pop actions, so we can't just use the FA graph.

To specify a PDA we have to specify

- a set $\Sigma$ of symbols that can labelled the transitions,
- a set $\mathcal{S}$ of states of the underlying FA,
- a start state $S \in \mathcal{S}$,
- a set $\mathcal{F} \subseteq \mathcal{S}$ of accepting states,
- a set $\Delta$ of symbols that can be pushed onto the stack,
- a set of actions.

The special end-of-symbol \$ is always put on the bottom of the stack. A string is accepted by the PDA if all the input has been read, so the current input symbol is \$, and the current state is an accepting state.

Sometimes acceptance is defined by saying that the current input symbol is \$ and the only symbol on the stack is \$. Both definitions allow exactly the same sets to be defined, although different PDAs need to be used for empty stack acceptance. We shall use the final acceptance state definition of acceptance, i.e. the input symbol is \$ and the current state is an accepting state.

It is easy to construct a PDA which accepts precisely the strings of the form $a^n b^n$ where $n \geq 1$. We have three states, $q$ which is the start state, $p$ and $r$, and $r$ is the accepting state. There are two input symbols $a, b$ and one stack symbol $\delta$. The actions are given in terms of the current state, input symbol and stack top, as follows:

- $(q, a, \$)$ – push $\delta$ onto the stack, goto state $q$ and read the next input symbol

  i.e. if the current state is $q$, the current input is $a$ and the current stack top is \$, we can push $\delta$ on to the stack and goto state $q$.

- $(q, a, \delta)$ – push $\delta$ onto the stack, goto state $q$ and read the next input symbol

  i.e. if the current state is $q$, the current input is $a$ and the current stack top is $\delta$, we can push $\delta$ on to the stack and goto state $q$.

- $(q, b, \delta)$ – pop $\delta$ off the stack, goto state $p$ and read the next input symbol

  i.e. if the current state is $q$, the current input is $b$ and the current stack top is $\delta$, we can pop $\delta$ off the stack and goto state $p$.

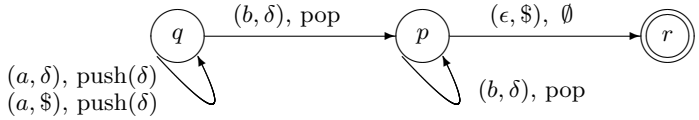
- $(p, b, \delta)$ – pop $\delta$ off the stack, goto state $p$ and read the next input symbol

  i.e. if the current state is $p$, the current input is $b$ and the current stack top is $\delta$, we can pop $\delta$ off the stack and goto state $p$.

- $(p, \epsilon, \$)$ – goto state $r$

  i.e. if the current state is $p$ and the current stack top is \$, we can goto state $r$.

In this case we can think of the PDA graphically, labelling the transitions with the input symbol to be read, the stack top required, and with the appropriate pop and push actions.




There are no push or pop actions associated with the transition from $p$ to $r$ so this is denoted by the empty set $\emptyset$.

Each time a symbol $a$ is read a $\delta$ is pushed onto the stack. Then each time a $b$ is read a $\delta$ is popped off the stack. If the same number of $b$s are read as $a$s then at the end the stack will have \$ on top and the string will be accepted.

The following sequence of moves shows the current state and current stack when the above PDA is run on the string $a^2 b^2$.

$$(q, (\$)) \underset{a}{\rightarrow} (q, (\$, \delta)) \underset{a}{\rightarrow} (q, (\$, \delta, \delta)) \underset{b}{\rightarrow} (p, (\$, \delta)) \underset{b}{\rightarrow} (p, (\$))$$

### 9.1.1 Formal definition of a PDA

Formally PDA actions are represented as a transition function $f$ from the set $\mathcal{S} \times (\Sigma \cup \{\epsilon\}) \times \Delta$ to the set of finite subsets of $\mathcal{S} \times \Delta^*$. So for a state $p$, an input symbol $a$ and a stack symbol $\delta$, we define $f(p, a, \delta)$ as a set of elements of the form $(r, (\delta_1, \ldots, \delta_m))$.

Then when the current state is $p$, the current input is $a$ and the current stack top is $\delta$, we choose an element $(r, (\delta_1, \ldots, \delta_m)) \in f(p, a, \delta)$. Then we pop $\delta$ off the stack, push $\delta_1, \ldots, \delta_m$ onto the stack, read the next input symbol and move to state $r$.

A string is accepted by PDA if it is accepted for some choices of elements. The set of accepted strings is the *language recognized by the PDA*.

Technically, every action pops the top symbol off the stack, so every action pops one symbol and pushes zero or more symbols. In our informal description we allowed an action to pop nothing, but we can always do this in the formal way by pushing back on the symbol we have just popped.

The above PDA for the language $\{a^n b^n \mid n \in \mathbb{P}\}$ is written formally as

$$\Sigma = \{a, b\}, \quad \mathcal{S} = \{q, p, r\}, \quad S = q, \quad \mathcal{F} = \{r\}, \quad \Delta = \{\delta, \$\}$$

$$\begin{aligned} f(q, a, \delta) &= \{(q, (\delta, \delta))\} \\ f(q, a, \$) &= \{(q, (\$, \delta))\} \\ f(q, b, \delta) &= \{(p, \epsilon)\} \\ f(p, b, \delta) &= \{(p, \epsilon)\} \\ f(p, \epsilon, \$) &= \{(r, \$)\} \end{aligned}$$

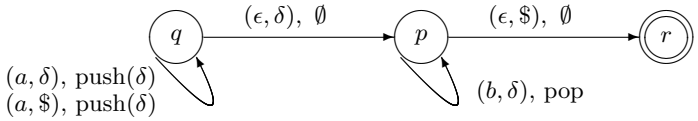
Note, we can perform an action without reading an input symbol, this is equivalent to traversing an $\epsilon$-arrow in an FA. Such actions are specified by writing $\epsilon$ instead of the input symbol for the second argument of the function $f$.

For example, the PDA

$$\Sigma = \{a, b\}, \quad \mathcal{S} = \{q, p, r\}, \quad S = q, \quad \mathcal{F} = \{r\}, \quad \Delta = \{\delta, \$\}$$

$$\begin{aligned} f(q, a, \delta) &= \{(q, (\delta, \delta))\} \\ f(q, a, \$) &= \{(q, (\$, \delta))\} \\ f(q, \epsilon, \delta) &= \{(p, \delta)\} \\ f(p, b, \delta) &= \{(p, \epsilon)\} \\ f(p, \epsilon, \$) &= \{(r, \$)\} \end{aligned}$$

also accepts precisely the set of strings $\{a^n b^n \mid n \in \mathbb{P}\}$. Graphically, the PDA can be thought of as




However, the first PDA for $\{a^n b^n \mid n \in \mathbb{P}\}$ is preferable to the second one because the first one is deterministic.

### 9.1.2 Deterministic and non-deterministic PDAs

**Definition** A PDA is *deterministic* if, for every state $p$, every input symbol $b$ and stack symbol $\gamma$, $f(p, b, \gamma)$ contains at most one element, and if, for some $c$, $f(p, c, \gamma)$ is not empty then $f(p, \epsilon, \gamma)$ is the empty set.

The point here is that for a deterministic PDA there is always at most one choice of action for a given state, input symbol and stack top. So for deterministic PDAs we

traverse the PDA using an input string and this traversal will result in acceptance if and only if the string is in the language defined by the PDA.

For a non-deterministic PDA it is possible for the traversal to stop without success even though the string is in the language. For example, for the second PDA for $\{a^n b^n \mid n \in \mathbb{N}\}$ we can traverse the PDA with the string $a^2b^2$ using the steps

$$(q, (\$)) \underset{a}{\rightarrow} (q, (\$, \delta)) \underset{\epsilon}{\rightarrow} (p, (\$, \delta))$$

At this point the next input symbol is $a$ but $f(p, a, \delta) = \emptyset$ so there are no further steps that can be carried out. As the input symbol is not \$, the string is not accepted.

The following is a straight-forward algorithm for determining whether or not a given string is accepted by a deterministic PDA.

```
set cs to the start state
set stack = $ and ct to $
set flag to 1
set ci to be the first input symbol

while  (ci is not $) and (flag==1) {
    if (p, ct) is in f(cs,ci,ct) {
          set cs = p
          set ci to be the next input symbol }
    else
       if (p, (ct,y)) is in f(cs,epsilon,ct) {
             set cs = p
             set ct = y
             push y onto stack }
       else
           if (p, epsilon) is in f(cs,epsilon,p) {
                 set cs = p
                 set ct = pop stack }
           else { set flag=0 }
}

if ( (flag = 1) and (cs is accepting) ) {accept the string}
else  {reject the string}
```

However, this algorithm does not work for non-deterministic PDAs because it may reject a string because it made a wrong choice at a point of non-determinism.

**Example** The following PDA accepts precisely the non-empty palindromes of even length on the letters $a$ and $b$. (That is strings of the form $\alpha\alpha^T$ where $\alpha^T$ is obtained by reversing the letters of $\alpha$.)

The idea is that as the elements of the first half of the string are read they are pushed onto the stack. Then as the second half of the string is read the symbols are popped off and matched to the input symbols. Thus in this example the stack symbols are the same as the input symbols.

$$\Sigma = \{a, b\}, \quad \mathcal{S} = \{q, p, r\}, \quad S = q, \quad \mathcal{F} = \{r\}, \quad \Delta = \{a, b, \$\}$$

$$\begin{aligned}
f(q,a,a) &= \{(q,(a,a)),(p,\epsilon)\}\\
f(q,a,b) &= \{(q,(b,a))\}\\
f(q,a,\$) &= \{(q,(\$,a))\}\\
f(q,b,a) &= \{(q,(a,b))\}\\
f(q,b,b) &= \{(q,(b,b)),(p,\epsilon)\}\\
f(q,b,\$) &= \{(q,(\$,b))\}\\
f(p,a,a) &= \{(p,\epsilon)\}\\
f(p,b,b) &= \{(p,\epsilon)\}\\
f(p,\epsilon,\$) &= \{(r,\$)\}
\end{aligned}$$

This PDA is non-deterministic because for some configurations there is a choice of action, for example when the current state is $q$ and the current input symbol and stack top are both $a$ we can use the action $(q,(a,a))$, pushing $a$ on to the stack and staying state $q$, or we can use the action $(p,\epsilon)$, popping the $a$ off the stack and moving to state $p$.

We can traverse this PDA with the string *aabbaa* using the steps

$$(q,\$) \underset{a}{\to} (q,(\$,a)) \underset{a}{\to} (q,(\$,a,a)) \underset{b}{\to} (q,(\$,a,a,b)) \underset{b}{\to} (p,(\$,a,a)) \underset{a}{\to} (p,(\$,a)) \underset{a}{\to} (p,\$) \underset{\epsilon}{\to} (r,\$)$$

Since, at the end, the input symbol is \$ and the current state is $r$, an accepting state, the string *aabbaa* is (correctly) accepted.

## Non-deterministic PDAs

There do exist algorithms which determine whether a given string is in accepted by any (non-deterministic) PDA. But none of these algorithms runs in linear time. The most common of these algorithms proceed by computing in parallel the different execution steps which can be taken at points of non-determinism. If this is not done carefully, the resulting algorithm can be of worst case exponential complexity and this is completely impractical. However, it is possible to construct algorithms which have worst case cubic complexity. A well known algorithm was given by Jay Earley in 1970, who was interested in constructing efficient parsers for natural language applications. An algorithm which computes in worst case cubic time whether or not a string is accepted by a given non-deterministic PDA was given by Lang in 1974.

The PDA for $\{a^n b^n \mid n \in \mathbb{P}\}$ at the start of this section is a deterministic PDA which corresponds to the non-deterministic one. However, there exist non-deterministic PDAs for which there is no corresponding deterministic PDA. In fact, we have the following theorem.

**Theorem 2** *The language* $\{a^n b^m \mid n \in \mathbb{N}, m = n \text{ or } m = 2n\}$ *is not the language accepted by any deterministic PDA.*

Regular expressions are used to define the words of a programming language, the sentences are defined using a *context free grammar*. Context free grammars correspond to pushdown automata in the same way that regular expressions correspond to finite state automata. The fact that there exist non-deterministic PDAs for which there is no corresponding deterministic PDA means that parsing, the recognition of languages generated by context free grammars, is a more difficult problem than lexical analysis, the recognition of languages generated by regular expressions. This has important implications for the design of compilers, and the constraints that this puts on grammars fundamentally influences the structure and style of modern programming languages. Pioneering work on parsing of programming languages was done by Don Knuth in the 1960s. Knuth defined a form of PDA called an LR(1) PDA and proved that any deterministic PDA is equivalent

to a some LR(1) PDA. We do not discuss context free grammars and the specification of programming language syntax in this course.

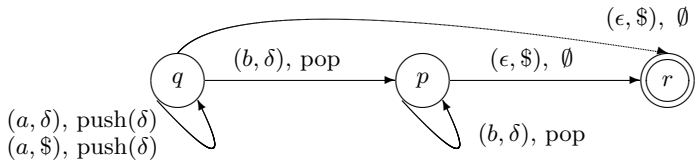
### 9.1.3 Examples

**Example** The following PDA accepts precisely the language $\{a^n b^n \mid n \in \mathbb{N}\}$ (this language includes the empty string, $\epsilon$).

$$\Sigma = \{a, b\}, \quad \mathcal{S} = \{q, p, r\}, \quad S = q, \quad \mathcal{F} = \{r\}, \quad \Delta = \{\delta, \$\}$$

$$\begin{aligned} f(q, a, \delta) &= \{(q, (\delta, \delta))\} \\ f(q, a, \$) &= \{(q, (\$, \delta))\} \\ f(q, \epsilon, \$) &= \{(r, \$)\} \\ f(q, b, \delta) &= \{(p, \epsilon)\} \\ f(p, b, \delta) &= \{(p, \epsilon)\} \\ f(p, \epsilon, \$) &= \{(r, \$)\} \end{aligned}$$

The corresponding graphical representation is




**Example** The following PDA accepts precisely the palindromes of odd length on the letters $a$ and $b$. (That is strings of the form $\alpha x \alpha^T$ where $\alpha^T$ is obtained by reversing the letters of $\alpha$ and $x$ is either $a$ or $b$.)

$$\Sigma = \{a, b\}, \quad \mathcal{S} = \{q, p, r\}, \quad S = q, \quad \mathcal{F} = \{r\}, \quad \Delta = \{a, b, \$\}$$

$$\begin{aligned} f(q, a, a) &= \{(q, (a, a)), (p, a)\} \\ f(q, a, b) &= \{(q, (b, a)), (p, b)\} \\ f(q, a, \$) &= \{(q, (\$, a)), (p, \$)\} \\ f(q, b, a) &= \{(q, (a, b)), (p, a)\} \\ f(q, b, b) &= \{(q, (b, b)), (p, b)\epsilon)\} \\ f(q, b, \$) &= \{(q, (\$, b)), (p, \$)\} \\ f(p, a, a) &= \{(p, \epsilon)\} \\ f(p, b, b) &= \{(p, \epsilon)\} \\ f(p, \epsilon, \$) &= \{(r, \$)\} \end{aligned}$$

This PDA is again non-deterministic. We can traverse this PDA with the string *abbba* using the steps

$$(q, \$) \underset{a}{\to} (q, (\$, a)) \underset{b}{\to} (q, (\$, a, b)) \underset{b}{\to} (p, (\$, a, b)) \underset{b}{\to} (p, (\$, a)) \underset{a}{\to} (p, \$) \underset{\epsilon}{\to} (r, \$)$$

Since, at the end, the input symbol is \$ and the current state is $r$ the string *abbba* is (correctly) accepted.

## 9.2 The Chomsky hierarchy

We have seen that regular expressions are not powerful enough to describe all sets, there is a set which is not definable with a regular expression. We have also seen that some sets not definable by a regular expression can be defined by a PDA.

It is thus reasonable to ask whether all sets can be defined by PDAs. The answer to this questions is no.

**Theorem 3** *The set* $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ *cannot be defined by any PDA.*

Classes of languages were studied by the linguist Noam Chomsky in the 1950s. He defined four classes of sets (languages) for which there exist finite specifications.

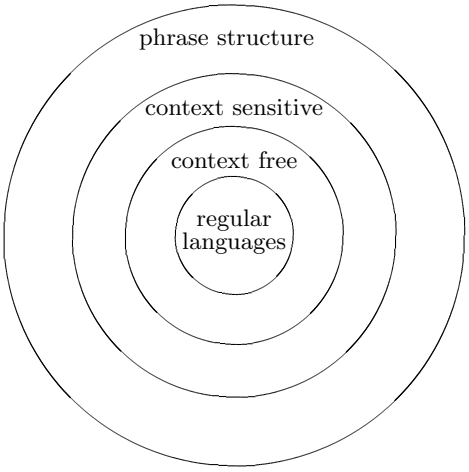
The first class is the sets which can be defined by regular expressions, the *regular languages*. The elements in a set in this class can be also described by a DFA.

The second class is the sets which can be the *context free languages*. The elements in a set in this class precisely that sets that can be described by a PDA.

The third class is the sets which can be defined by grammars called *context sensitive grammars*. Most languages that occur in practice are context sensitive, but there are some sets which are not context sensitive but whose elements can be computed.

The fourth class is the sets which can be defined by grammars called *phrase structure grammars*.

Each class of languages is a proper subclass of the next one. This is referred to as the *Chomsky Hierarchy*.




Phrase structure languages are precisely the sets that can be described with a Turing machine, the topic of the rest of this chapter.

## 9.3 Machines Can Never Be Enough

In this section we look briefly at some of the issues which lie at the limits of computability. It is important that computer scientists know that it is impossible for computers, or finitely describable machines of any type, to be able to do everything that is mathematically possible. In fact there exist sets for which it is impossible for any computer to compute the elements of the set. It is thus impossible for computers, or finitely describable machines of any type, to compute every function $f$ of the type $f :\rightarrow$.

### 9.3.1 Turing machines and Church's thesis

It is not possible to give a formal definition of what we mean by saying that something is mechanically calculable, or what we mean by a machine or a computer. Intuitively by a machine or a computer we mean some sort of process whose behaviour in terms of input and output can be described using a finite specification. This does not mean that the

calculations the machine performs must always be finite, but the description of how the machine can behave must be finite.

The famous mathematician and computer scientist Alan Turing spent some time searching for a definition of a machine, and eventually he produced the definition of what is now known as a Turing machine. We cannot prove that anything that we recognise as a machine, or a mechanical process, can be represented by a Turing machine because we cannot formally define the concept 'anything that we recognise as a machine, or a mechanical process'. However, every machine that anyone has ever thought of has been shown to be representable as a Turing machine, including what initially appear to be more powerful or more general versions of Turing machines themselves.

It is generally accepted that anything that could be considered to be a machine can be represented by a Turing machine, and this belief is formally known as Church's Thesis: if a function $f :\rightarrow$ is computable by any kind of finite machine, it is computable by Turing machine.

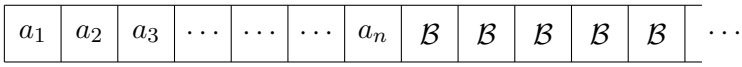
Church's Thesis implies that every machine that can be built is a Turing machine. Thus for the purposes of reasoning about what can and cannot be computed by some machine, we reason about what can and cannot be computed by a Turing machine.

Perhaps the most surprising thing about Turing machines is that their structure is relatively simple, their definition is similar to that of pushdown automata.

A Turing machine has a set of states and a tape from which it can read and to which it can write. The tape has a left hand end but is infinite to the right and all but finitely many of the symbols on the tape are the special blank symbol $\mathcal{B}$. The machine looks at the current state and the symbol on the current position of the tape and then executes an action on the basis of these two values. The action consists of writing a symbol to the current tape position, moving the tape position one place to the left or to the right, and moving to a specified state.

**Definition** A Turing machine is a 7-tuple $(\mathcal{S}, \Sigma, \Delta, \mathrm{f}, S, \mathcal{B}, \mathcal{F})$ where
$\mathcal{S}$ is a finite set of states and $S \in \mathcal{S}$ is the start state
$\Delta$ is a finite set of tape symbols, and $\mathcal{B} \in \Delta$ is a special blank symbol
$\Sigma \subseteq \Delta$ is a finite set of input symbols
$f$ is a (partial) function from $\mathcal{S} \times \Delta$ to $\mathcal{S} \times \Delta \times \{L, R\}$ which assigns actions to state, tape configurations
$\mathcal{F} \subseteq \mathcal{S}$ is a set of accepting states.

A Turing machine begins with a tape on which there are written finitely many (possibly 0) symbols from $(\Delta \backslash \{\mathcal{B}\})$, the remaining entries on the tape are all $\mathcal{B}$.




The initial input position of the tape is the first (left-most) position, reading the symbol $x_1$ say, and the state is $S$. If $f(S, x_1) = (t, x_2, R)$ then the machine replaces $x_1$ with $x_2$ on the tape, moves the tape position one place to the right and makes $t$ the current state. This process continues until either the machine moves to an accepting state or there is no move associated with the current state and input symbol. In the former case the initial string written on the tape is said to be *accepted* by the Turing machine. If the machine reaches a point where there is no action, or if the action would result in the pointer moving to the left of the end of the tape, the initial string written on the tape is said to be *rejected*.

The language defined by a Turing machine is the set of strings that it accepts.

The Turing machines that we have defined are deterministic because there is at most one choice of action for each state and input symbol. Thus we might believe that a Turing

machine can deterministically calculate the language it defines. Since we have claimed that Turing machines can be written for any computational process, this would imply that all such processes could be carried out!

The fly in the ointment is that a Turing machine may not always halt. A Turing machine is said to *halt* if either it moves into an accepting state, or if there is no action associated with the current state and input symbol.

In fact Turing himself proved that it is not possible to decide which Turing machines halt. There is no algorithm (or machine) which when given any Turing machine will always be able to compute whether or not the Turing machine halts on all its inputs.

Of course, there are many Turing machines for which we can show the machine always halts on all its inputs, and there are some Turing machines which we can show never halt on certain inputs. But there is no algorithm which can decide the result for all machines and all inputs.

There are many other definitions of machines which are similar to Turing machines and which seem, at first sight, to be more general. For example, we can allow the tape to be infinite in both directions. Also we can allow more than one choice of action to be associated with a given state and input symbol (a non-deterministic Turing machine).

It is a remarkable fact, and one which significantly supports Church's Thesis, that all the extensions of Turing machines which have been proposed turn out to the equivalent to the original definition. In other words, for any formal machine that has ever been proposed there is an equivalent Turing machine.

### 9.3.2 Non-computability

The sets which are languages defined by Turing machines are called the *recursively enumerable sets*.

The point about recursively enumerable sets is that there is an algorithm which lists all the elements in the set. Thus if a set $L$ is recursively enumerable (r.e.) and we want to know if $x \in L$ we set the algorithm going and in finite time the element $x$ will appear on the list of outputs.

This does not mean, however, that we can decide whether or not $x \in L$. If $x \in L$ then it will eventually appear on the list. But if $x \notin L$ we can't know this. The fact that it has not appeared on the list so far does not mean that it will not appear later.

However, if both the set $L$ and its complement are r.e. then we can decide for any given $x$ whether $x \in L$. This is because we can set both algorithms, the one which lists $L$ and the one which lists its complement, going and in finite time the element $x$ will appear on one list or the other.

Sets $L$ which are r.e. and whose complement is also r.e. are called *recursive sets*. It is the case that the recursive sets are precisely the languages of the Turing machines which always eventually halt on all inputs.

Recall the Chomsky hierarchy described above. The most general class of languages in the hierarchy is the phrase structure class, and it is a theorem that the phrase structure languages are precisely the r.e. sets.

The question arises as to whether all sets can be defined by Turing machines. In other words are all sets r.e. If this is not true then, by Church's Thesis, there exist sets whose elements cannot be computed!

The proof that there exist sets which are not r.e. is the same as the proof that the real numbers are not countable. We use a diagonalisation argument.

First we note that since every Turing machine has a finite set of states and a finite set of tape symbols, and thus there are only finitely many functions $f$ on $\mathcal{S} \times \Delta$, we can form a (infinite) list of all the possible Turing machines:

$$T_1, T_2, T_3, \ldots$$

Consider all the functions $f : \mathbb{N} \to \mathbb{N}$. Let us suppose that the sub-list of Turing machines which calculate integer functions is

$$T_{i_1}, T_{i_2}, T_{i_3}, \ldots$$

Now define a function $g : \mathbb{N} \to \mathbb{N}$ as follows:

If, for the integer function $f_{i_m}$ calculated by $T_{i_m}$, we have $f_{i_m}(m) = 0$ then let $g(m) = 1$, otherwise let $g(m) = 0$.

Then we have that $g \neq f_{i_k}$ for any value of $k$, thus $g$ cannot be calculated by any Turing machine.

Put informally, there are only countably many Turing machines, but there are uncountably many integer functions, so there are too many functions for them all to be computable.

For a function $f : \mathbb{N} \to \mathbb{N}$ define the set $X_f = \{(n, f(n)) \mid n \in \mathbb{N}\}$. If $f$ is a function which cannot be computed using a Turing machine then $X_f$ cannot be the language of a Turing machine, i.e. $X_f$ is not r.e.

## 9.4 Look up on the WEB

**Alan Turing** Born in London in 1912, Turing studied mathematics at Cambridge. During the second World War Turing worked on code breaking at Bletchley Park. He wrote many papers on mathematics, logic and computability, but perhaps he is best known in the academic world for his paper on the Entscheidungsproblem in which he proves the undecidability of the halting problem.

**Alonzo Church** Born in the USA in 1903, in addition to formulating Church's Thesis, Church invented the Lambda Calculus, a logical formalism which is important in the study of the semantics of programming languages. Turing was a student of Church's in Princeton for a while, and so was Kleene.

**Don Knuth** Born in 1938 in America, Knuth started publishing papers while he was still an undergraduate. In addition to his fundamental contribution to parsing theory, Knuth create the TeX type setting system which completely revolutionised mathematical and scientific publishing.

**Noam Chomsky** Born in America in 1928, Chomsky worked for most of his career at MIT in Boston. He wrote many papers in linguistics and in politics and philosophy. Chomsky is one of the most frequently cited authors who is still alive.