Theory in Programming Practice

Jayadev Misra
The University of Texas at Austin
Austin, Texas 78712, USA
email: misra@cs.utexas.edu

December 2004

Preface

"The distance between theory and practice is always smaller in theory than in practice." — Marshall T. Rose

Computer Programming has been, largely, an intuitive activity. Programmers are taught to understand programming in operational terms, i.e., how a computer executes a program. As the field has matured, we see many effective theories for designing and reasoning about computer programs in specific domains. Such theories reduce the mental effort needed to design a product, and they are as indispensable for their domains as calculus is for solving scientific and engineering problems. I am attracted to effective theories primarily because they save labor (for the human and the computer), and secondarily because they give us better assurance about the properties of programs.

The original inspiration to design a computer science course which illustrates the applications of effective theories in practice came from Elaine Rich and J Moore. I prepared a set of notes and taught the course in the Spring and Fall of 2003. The choice of topics and the style of presentation are my own. I have made no effort to be comprehensive.

Greg Plaxton has used my original notes and made extensive revisions; I am grateful to him. I am also grateful to the graduate teaching assistants, especially Thierry Joffrain, for helping me revise the notes.

Austin, Texas J. Misra
December 2004

Contents

Chapter 1

Text Compression

1.1 Introduction

Data compression is useful and necessary in a variety of applications. These applications can be broadly divided into two groups: transmission and storage. Transmission involves sending a file, from a *sender* to a *receiver*, over a channel. Compression reduces the number of bits to be transmitted, thus making the transmission process more efficient. Storing a file in a compressed form typically requires fewer bits, thus utilizing storage resources (including main memory itself) more efficiently.

Data compression can be applied to any kind of data: text, image (such as fax), audio and video. A 1-second video without compression takes around 20 megabytes (i.e., 170 megabits) and a 2-minute CD-quality uncompressed music (44,100 samples per second with 16 bits per sample) requires more than 84 megabits. Impressive gains can be made by compressing video, for instance, because successive frames are very similar to each other in their contents. In fact, real-time video transmission would be impossible without considerable compression. There are several new applications that generate data at prodigious rates; certain earth orbiting satellites create around half a terabyte (10^{12}) of data per day. Without compression there is no hope of storing such large files in spite of the impressive advances made in storage technologies.

Lossy and Lossless Compression Most data types, except text, are compressed in such a way that a very good approximation, but not the exact content, of the original file can be recovered by the receiver. For instance, even though the human voice can range up to 20kHz in frequency, telephone transmissions retain only up to about 5kHz.[1] The voice that is reproduced at the receiver's end is a close approximation to the real thing, but it is not exact. Try lis-

[1] A famous theorem, known as the *sampling theorem*, states that the signal must be sampled at twice this rate, i.e., around 10,000 times a second. Typically, 8 to 16 bits are produced for each point in the sample.

tening to your favorite CD played over a telephone line. Video transmissions often sacrifice quality for speed of transmission. The type of compression in such situations is called *lossy*, because the receiver cannot exactly reproduce the original contents. For analog signals, all transmissions are lossy; the degree of loss determines the quality of transmission.

Text transmissions are required to be *lossless*. It will be a disaster to change even a single symbol in a text file.[2] In this note, we study several lossless compression schemes for text files. Henceforth, we use the terms *string* and *text file* synonymously.

Error detection and correction can be applied to uncompressed as well as compressed strings. Typically, a string to be transmitted is first compressed and then encoded for errors. At the receiver's end, the received string is first decoded (error detection and correction are applied to recover the compressed string), and then the string is decompressed.

What is the typical level of compression? The amount by which a text string can be compressed depends on the string itself. A repetitive lyric like "Old McDonald had a farm" can be compressed significantly, by transmitting a single instance of a phrase that is repeated.[3] I compressed a postscript file of 2,144,364 symbols to 688,529 symbols using a standard compression algorithm, `gzip`; so, the compressed file is around 32% of the original in length. I found a web site[4] where *The Adventures of Tom Sawyer*, by Mark Twain, is in uncompressed form at 391 Kbytes and compressed form (in zip format) at 172 Kbytes; the compressed file is around 44% of the original.

1.2 A Very Incomplete Introduction to Information Theory

Take a random string of symbols over a given alphabet; imagine that there is a source that spews out these symbols following some probability distribution over the alphabet. If all symbols of the alphabet are equally probable, then you can't do any compression at all. However, if the probabilities of different symbols are non-identical —say, over a binary alphabet "0" occurs with 90% frequency and "1" with 10%— you may get significant compression. This is because you are likely to see runs of zeros more often, and you may encode such runs using short bit strings. A possible encoding, using 2-bit blocks, is: 00 for 0, 01 for 1, 10 for 00 and 11 for 000. We are likely to see a large number of "000" strings which would be compressed by one bit, whereas for encoding "1" we lose a bit.

[2] There are exceptions to this rule. In some cases it may not matter to the receiver if extra white spaces are squeezed out, or the text is formatted slightly differently.

[3] Knuth [27] gives a delightful treatment of a number of popular songs in this vein.

[4] `http://www.ibiblio.org/gutenberg/cgi-bin/sdb/t9.cgi/t9.cgi?entry=74&full=yes&ftpsite=http://www.ibiblio.org/gutenberg/`

In 1948, Claude E. Shannon [42] published "A Mathematical Theory of Communication", in which he presented the concept of *entropy*, which gives a quantitative measure of the compression that is possible. I give below an extremely incomplete treatment of this work.

Consider a finite alphabet; it may be binary, the Roman alphabet, all the symbols on your keyboard, or any other finite set. A random source outputs a string of symbols from this alphabet; it has probability p_i of producing the ith symbol. Productions of successive symbols are independent, that is, for its next output, the source selects a symbol with the given probabilities independent of what it has produced already. The *entropy*, h, of the alphabet is given by

$$h \;=\; -\sum_i p_i \;(\log p_i)$$

where log stands for logarithm to base 2. Shannon showed that you need at least h bits on the average to encode each symbol of the alphabet; i.e., for lossless transmission of a (long) string of n symbols, you need at least nh bits. And, it is possible to transmit at this rate!

To put this theory in concrete terms, suppose we have the binary alphabet where the two symbols are equiprobable. Then,

$$\begin{aligned} h &= -0.5 \times (\log 0.5) - 0.5 \times (\log 0.5) \\ &= -\log 0.5 \\ &= 1 \end{aligned}$$

That is, you need 1 bit on the average to encode each symbol, so you cannot compress such strings at all! Next, suppose the two symbols are not equiprobable; "0" occurs with probability 0.9 and "1" with 0.1. Then,

$$\begin{aligned} h &= -0.9 \times (\log 0.9) - 0.1 \times (\log 0.1) \\ &= 0.469 \end{aligned}$$

The text can be compressed to less than half its size. If the distribution is even more lop-sided, say 0.99 probability for "0" and 0.01 for "1", then $h = 0.080$; it is possible to compress the file to 8% of its size.

Exercise 1

Show that for an alphabet of size m where all symbols are equally probable, the entropy is $\log m$. □

Next, consider English text. The source alphabet is usually defined as the 26 letters and the space character. There are then several models for entropy. The zero-order model assumes that the occurrence of each character is equally likely. Using the zero-order model, the entropy is $h = \log 27 = 4.75$. That is, a string of length n would have no less than $4.75 \times n$ bits.

The zero-order model does not accurately describe English texts: letters occur with different frequency. Six letters — 'e', 't', 'a', 'o', 'i', 'n'— occur over half the time; see Tables 1.1 and 1.2. Others occur rarely, such as 'q' and 'z'. In

Letter	Frequency	Letter	Frequency	Letter	Frequency	Letter	Frequency
a	0.08167	b	0.01492	c	0.02782	d	0.04253
e	0.12702	f	0.02228	g	0.02015	h	0.06094
i	0.06966	j	0.00153	k	0.00772	l	0.04025
m	0.02406	n	0.06749	o	0.07507	p	0.01929
q	0.00095	r	0.05987	s	0.06327	t	0.09056
u	0.02758	v	0.00978	w	0.02360	x	0.00150
y	0.01974	z	0.00074				

Table 1.1: Frequencies of letters in English texts, alphabetic order

Letter	Frequency	Letter	Frequency	Letter	Frequency	Letter	Frequency
e	0.12702	t	0.09056	a	0.08167	o	0.07507
i	0.06966	n	0.06749	s	0.06327	h	0.06094
r	0.05987	d	0.04253	l	0.04025	c	0.02782
u	0.02758	m	0.02406	w	0.02360	f	0.02228
g	0.02015	y	0.01974	p	0.01929	b	0.01492
v	0.00978	k	0.00772	j	0.00153	x	0.00150
q	0.00095	z	0.00074				

Table 1.2: Frequencies of letters in English texts, descending order

the first-order model, we assume that each symbol is statistically independent (that is, the symbols are produced independently) but we take into account the probability distribution. The first-order model is a better predictor of frequencies and it yields an entropy of 4.219 bits/symbol. For a source alphabet including the space character, a traditional value is 4.07 bits/symbol with this model.

Higher order models take into account the statistical dependence among the letters, such as that 'q' is almost always followed by 'u', and that there is a high probability of getting an 'e' after an 'r'. A more accurate model of English yields lower entropy. The third-order model yields 2.77 bits/symbol. Estimates by Shannon [43] based on human experiments have yielded values as low as 0.6 to 1.3 bits/symbol.

Compression Techniques from Earlier Times Samuel Morse developed a code for telegraphic transmissions in which he encoded the letters using a binary alphabet, a dot (·) and a dash (–). He assigned shorter codes to letters like 'e'(·) and 'a'(· –) that occur more often in texts, and longer codes to rarely-occurring letters, like 'q'(– – · –) and 'j'(· – – –).

The Braille code, developed for use by the blind, uses a 2×3 matrix of dots where each dot is either flat or raised. The 6 dots provide $2^6 = 64$ possible combinations. After encoding all the letters, the remaining combinations are assigned to frequently occurring words, such as "and" and "for".

Symbol	Prob.	C1	avg. length	C2	avg. length	C3	avg. length
a	0.05	00	$0.05 \times 2 = 0.1$	00	$0.05 \times 2 = 0.1$	000	$0.05 \times 3 = 0.15$
c	0.5	0	$0.5 \times 1 = 0.5$	01	$0.5 \times 2 = 1.0$	1	$0.5 \times 1 = 0.5$
g	0.4	1	$0.4 \times 1 = 0.4$	10	$0.4 \times 2 = 0.8$	01	$0.4 \times 2 = 0.8$
t	0.05	11	$0.05 \times 2 = 0.1$	11	$0.05 \times 2 = 0.1$	001	$0.05 \times 3 = 0.15$
			exp. length = 1.1		2.0		1.6

Table 1.3: Three different codes for $\{a, c, g, t\}$

1.3 Huffman Coding

We are given a set of symbols and the probability of occurrence of each symbol in some long piece of text. The symbols could be $\{0, 1\}$ with probability 0.9 for 0 and 0.1 for 1, Or, the symbols could be $\{a, c, g, t\}$ from a DNA sequence with appropriate probabilities, or Roman letters with the probabilities shown in Table 1.1. In many cases, particularly for text transmissions, we consider frequently occurring words —such as "in", "for", "to"— as symbols. The problem is to devise a *code*, a binary string for each symbol, so that (1) any encoded string can be decoded (i.e., the code is *uniquely decodable*, see below), and (2) the expected code length —probability of each symbol times the length of the code assigned to it, summed over all symbols— is minimized.

Example Let the symbols $\{a, c, g, t\}$ have the probabilities 0.05, 0.5, 0.4, 0.05 (in the given order). We show three different codes, C1, C2 and C3, and the associated expected code lengths in Table 1.3.

Code C1 is not uniquely decodable because cc and a will both be encoded by 00. Code C2 encodes each symbol by a 2 bit string; so, it is no surprise that the expected code length is 2.0 in this case. Code C3 has variable lengths for the codes. It can be shown that C3 is optimal, i.e., it has the minimum expected code length.

1.3.1 Uniquely Decodable Codes and Prefix Codes

We can get low expected code length by assigning short codewords to every symbol. If we have n symbols we need n distinct codewords. But that is not enough. As the example above shows, it may still be impossible to decode a piece of text unambiguously. A code is *uniquely decodable* if every string of symbols is encoded into a different string.

A *prefix code* is one in which no codeword is a prefix of another.[5] The codewords 000, 1, 01, 001 for $\{a, c, g, t\}$ constitute a prefix code. A prefix code is uniquely decodable: if two distinct strings are encoded identically, either their first symbols are identical (then, remove their first symbols, and repeat this step

[5]String s is a prefix of string t if $t = s +\!\!+ x$, for some string x, where $+\!\!+$ denotes concatenation.

until they have distinct first symbols), or the codeword for one first symbol is a prefix of the other first symbol, contradicting that we have a prefix code.

It can be shown —but I will not show it in these notes— that there is an optimal uniquely decodable code which is a prefix code. Therefore, we can limit our attention to prefix codes only, which we do in the rest of this note.

A prefix code can be depicted by a labeled binary tree, as follows. Each leaf is labeled with a symbol (and its associated probability), a left edge by 0 and a right edge by 1. The codeword associated with a symbol is the sequence of bits on the path from the root to the corresponding leaf. See Figure 1.1 for a prefix code for $\{a, c, g, t\}$ which have associated probabilities of 0.05, 0.5, 0.4, 0.05 (in the given order).

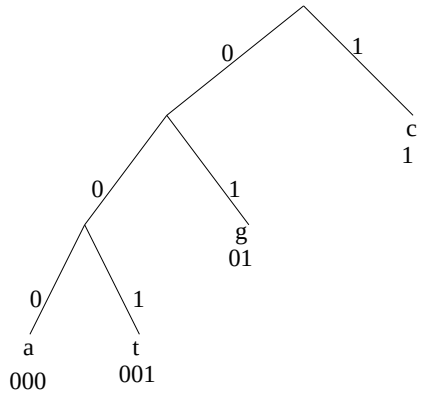


Figure 1.1: Prefix code for $\{a, c, g, t\}$

The length of a codeword is the corresponding pathlength. The *weighted pathlength* of a leaf is the probability associated with it times its pathlength. The *expected code length* is the sum of the weighted pathlengths over all leaves. Henceforth, the expected code length of a tree will be called its *weight*, and a tree is *best* if its weight is minimum. Note that there may be several best trees for the given probabilities.

Since the symbols themselves play no role —the probabilities identify the associated symbols— we dispense with the symbols and work with the probabilities only. Since the same probability may be associated with two different symbols, we have a bag, i.e., a multiset, of probabilities. Also, it is immaterial that the bag elements are probabilities; the algorithm applies to any bag of nonnegative numbers. We use the set notation for bags below.

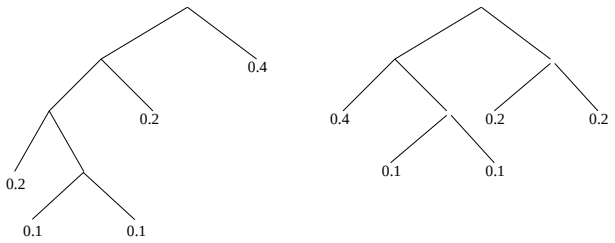
Exercise 2

Try to construct the best tree for the following values $\{1,\ 2,\ 3,\ 4,\ 5,\ 7,\ 8\}$. The weight of the best tree is 78. □

Remark: In a best tree, there is no dangling leaf; i.e., each leaf is labeled with a distinct symbol. Therefore, every internal node (i.e., nonleaf) has exactly two children. Such a tree is called a *full binary tree*.

Exercise 3

Show two possible best trees for the alphabet {0, 1, 2, 3, 4} with probabilities {0.2, 0.4, 0.2, 0.1, 0.1}. The trees should not be mere rearrangements of each other through reflections of subtrees. □



1.3.2 Constructing An Optimal Prefix Code

Huffman has given an extremely elegant algorithm for constructing a best tree for a given set of symbols with associated probabilities.[6]

The optimal prefix code construction problem is: given a bag of nonnegative numbers, construct a best tree. That is, construct a binary tree and label its leaves by the numbers from the bag so that the weight, i.e., the sum of the weighted pathlengths to the leaves, is minimized.

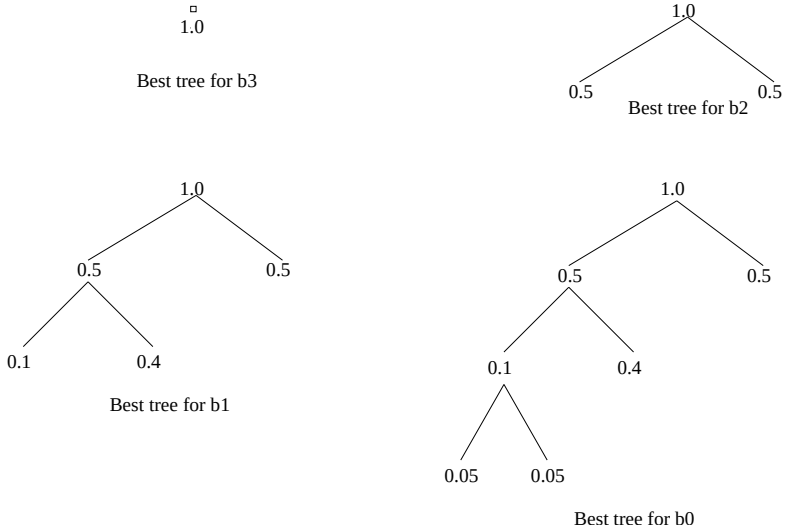
The Huffman Algorithm If bag b has a single number, create a tree of one node, which is both a root and a leaf, and label the node with the number. Otherwise (the bag has at least two numbers), let u and v be the two smallest numbers in b, not necessarily distinct. Let $b' = b - \{u, v\} \cup \{u + v\}$, i.e., b' is obtained from b by replacing its two smallest elements by their sum. Construct a best tree for b'. There is a leaf node in the tree labeled $u + v$; expand this node to have two children that are leaves and label them with u and v.

Illustration of Huffman's Algorithm Given a bag $\{0.05,\ 0.5,\ 0.4,\ 0.05\}$, we obtain successively

$b_0 = \{0.05,\ 0.5,\ 0.4,\ 0.05\}$	, the original bag
$b_1 = \{0.1,\ 0.5,\ 0.4\}$	, replacing $\{0.05,\ 0.05\}$ by their sum
$b_2 = \{0.5,\ 0.5\}$	, replacing $\{0.1,\ 0.4\}$ by their sum
$b_3 = \{1.0\}$	, replacing $\{0.5,\ 0.5\}$ by their sum

The trees corresponding to these bags are shown below:

[6] I call an algorithm elegant if it is easy to state and hard to prove correct.



1.3.3 Proof of Correctness

We prove that Huffman's algorithm yields a best tree.

Lemma 1: Let u and v be two smallest values in bag b. Then, there is a best tree for b in which u and v are siblings.

Proof: Consider any best tree for b. Let U and V be the pathlengths to u and v, respectively. Without loss in generality, assume that $U \geq V$. Let the sibling of u be x. We show that exchanging x and v does not increase the weighted pathlength; so, the resulting tree is also a best tree in which u and v are siblings. (Note: There is no need to consider $v = x$ as a special case. Then u and v are siblings and exchanging v and x has no effect.)

To compare the weighted pathlengths of the two trees, before and after the exchange, we need only compare the sum of the weighted pathlengths to x and v in both trees, because no other pathlength is affected by the exchange. In the original tree, the sum of the weighted pathlengths to x and v is $xU + vV$; after the exchange it is $xV + vU$. We show that $xU + vV \ \geq \ xV + vU$; that is, the weighted pathlength does not increase due to the exchange.

From the given assumptions,

$$
\begin{array}{ll}
 & U \geq V \text{ and } v \leq x \\
\Rightarrow & \{x - v \geq 0. \text{ Multiply both sides of } U \geq V \text{ by } x - v\} \\
 & (x - v)U \geq (x - v)V \\
\Rightarrow & \{\text{Arithmetic}\} \\
 & xU + vV \ \geq \ xV + vU
\end{array}
$$

□

The next theorem shows that Huffman's algorithm constructs a best tree.

Theorem: Given is a bag b. Let u and v be two smallest values in b. And, $b' = b - \{u, v\} \cup \{u + v\}$. There is a best tree T for b such that

1. u and v are siblings in T, and
2. T' is a best tree for b', where T' is all of T except the two nodes u and v; see Figure 1.2.

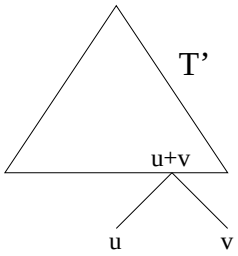


Figure 1.2: The entire tree is T for b; its upper part is T' for b'.

Proof: From the previous lemma, there is a best tree T for b in which u and v are siblings. This proves part 1.

Proof of part 2: Let the pathlength to the leaf $u + v$ in T' be n. Then, the pathlengths to u and v in T are $n + 1$. The weighted pathlength of T, $W(T)$, is obtained from $W(T')$ by replacing the term $n(u + v)$ in the sum by $(n + 1)u + (n + 1)v$, i.e.,

$$W(T) = W(T') + (u + v)$$

If T' is not a best tree for b' then there is a better tree, say S, for b'. Replace T' by S in T to create a better tree than T for b; its weighted pathlength is $W(S)+(u+v)$, which is lower than $W(T')+(u+v)$, i.e., $W(T)$. This contradicts the assumption that T is a best tree for b. □

This theorem justifies the Huffman algorithm.

Exercise 4

1. What is the structure of the Huffman tree for 2^n, $n \geq 0$, equiprobable symbols?
2. Show that the tree corresponding to an optimal prefix code is a *full* binary tree.
3. In a best tree, consider two nodes labeled x and y, and let the corresponding pathlengths be X and Y, respectively. Show that

$$x < y \;\Rightarrow\; X \geq Y$$

4. Prove or disprove (in the notation of the previous exercise)

   $x \leq y \Rightarrow X \geq Y$, and
   $x = y \Rightarrow X \geq Y$

5. Consider the first n fibonacci numbers (start at 1). What is the structure of the tree constructed by Huffman's algorithm on these values?

6. Give a 1-pass algorithm to compute the weight of the optimal tree.

7. Show that the successive values computed during execution of Huffman's algorithm (by adding the two smallest values) are nondecreasing.

8. (Research) As we have observed, there may be many best trees for a bag. We may wish to find the *very best* tree that is a best tree in which the maximum pathlength to any node is as small as possible. The following procedure achieves this: whenever there is a tie in choosing values, always choose an original value rather than a value obtained through combinations of previous values. Show the correctness of this method and also that it minimizes the sum of the pathelengths among all best trees. See Knuth [25], Section 2.3.4.5, page 404. □

How Good is Huffman Coding? We know from Information theory (see Section 1.2) that it is not possible to construct code whose weight is less than the entropy, but it is possible to find codes with this value (asymptotically). It can be shown that for any alphabet whose entropy is h, the Huffman code with weight H satisfies:

$$h \leq H < h + 1$$

So, Huffman coding comes extremely close to the predicted theoretical optimum if h is reasonably large.

However, in another sense, Huffman coding leaves much to be desired. The probabilities are very difficult to estimate if you are compressing something other than standard English novels. How do you get the frequencies of symbols in a postscript file? And, which ones should we choose as symbols in such a file? The latter question is very important because files tend to have bias toward certain phrases, and we can compress much better if we choose those as our basic symbols.

The Lempel-Ziv code, described in the following section addresses some of these issues.

1.3.4 Implementation

During the execution of Huffman's algorithm, we will have a bag of elements where each element holds a value and it points to either a leaf node —in case it represents an original value— or a subtree —if it has been created during the run of the algorithm. The algorithm needs a data structure on which the

following operations can be performed efficiently: (1) remove the element with the smallest value and (2) insert a new element. In every step, operation (1) is performed twice and operation (2) once. The creation of a subtree from two smaller subtrees is a constant-time operation, and is left out in the following discussion.

A priority queue supports both operations. Implemented as a heap, the space requirement is $O(n)$ and each operation takes $O(\log n)$ time, where n is the maximum number of elements. Hence, the $O(n)$ steps of Huffman's algorithm can be implemented in $O(n \log n)$ time.

There is an important special case in which the algorithm can be implemented in linear time. Suppose that the initial bag is available as a sorted list. Then, each operation can be implemented in constant time. Let *leaf* be the list of initial values sorted in ascending order. Let *nonleaf* be the list of values generated in sequence by the algorithm (by summing the two smallest values in *leaf* $\cup$ *nonleaf*).

The important observation is that

- (monotonicity) *nonleaf* is an ascending sequence.

You are asked to prove this in part 7 of the exercises in Section 1.3.3.

This observation implies that the smallest element in *leaf* $\cup$ *nonleaf* at any point during the execution is the smaller of the two items at the heads of *leaf* and *nonleaf*. That item is removed from the appropriate list, and the monotonicity property is still preserved. An item is inserted by adding it at the tail end of *nonleaf*, which is correct according to monotonicity.

It is clear that *leaf* is accessed as a list at one end only, and *nonleaf* at both ends, one end for insertion and the other for deletion. Therefore, *leaf* may be implemented as a stack and *nonleaf* as a queue. Each operation then takes constant time, and the whole algorithm runs in $O(n)$ time.

1.4 Lempel-Ziv Coding

As we have noted earlier, Huffman coding achieves excellent compression when the frequencies of the symbols can be predicted, and when we can identify the interesting symbols. In a book, say *Hamlet*, we expect the string *Ophelia* to occur quite frequently, and it should be treated as a single symbol. Lempel-Ziv coding does not require the frequencies to be known a priori. Instead, the sender scans the text from left to right identifying certain strings (henceforth, called *words*) that it inserts into a *dictionary*. Let me illustrate the procedure when the dictionary already contains the following words. Each word in the dictionary has an *index*, simply its position.

index	word
0	$\langle\rangle$
1	t
2	a
3	ta

index	word	transmission
0	$\langle\rangle$	none
1	t	$(0, t)$
2	a	$(0, a)$
3	c	$(0, c)$
4	ca	$(3, a)$
5	g	$(0, g)$
6	ta	$(1, a)$
7	cc	$(3, c)$
8	ag	$(2, g)$
9	tac	$(6, c)$
10	cac	$(4, c)$
11	$ta\#$	$(6, \#)$

Table 1.4: Transmission of *taccagtaccagtaccacta#* using Lempel-Ziv Code

Suppose the remaining text to be transmitted is *taaattaaa*. The sender scans this text from left until it finds a string that is *not* in the dictionary. In this case, t and ta are in the dictionary, but taa is not in the dictionary. The sender adds this word to the dictionary, and assigns it the next higher index, 4. Also, it transmits this word to the receiver. But it has no need to transmit the whole word (and, then, we will get no compression at all). The prefix of the word excluding its last symbol, i.e., ta, is a dictionary entry (remember, the sender scans the text just one symbol beyond a dictionary word). Therefore, it is sufficient to transmit $(3, a)$, where 3 is the index of ta, the prefix of taa that is in the dictionary, and a is the last symbol of taa.

The receiver recreates the string taa, by loooking up the word with index 3 and appending a to it, and then it appends taa to the text it has created already; also, it updates the dictionary with the entry

index	word
4	taa

Initially, the dictionary has a single word, the empty string, $\langle\rangle$, as its only (0th) entry. The sender and receiver start with this copy of the dictionary and the sender continues its transmissions until the text is exhausted. To ensure that the sender can always find a word which is not in the dictionary, assume that the end of the file, written as #, occurs nowhere else in the string.

Example Consider the text *taccagtaccagtaccacta#*. The dictionary and the transmissions are shown in Table 1.4. □

It should be clear that the receiver can update the dictionary and recreate the text from the given transmissions. Therefore, the sequence of transmissions constitutes the compressed file. In the small example shown above, there is hardly any compression. But for longer files with much redundancy, this scheme

achieves excellent results. Lempel-Ziv coding is asymptotically optimal, i.e., as the text length tends to infinity, the compression tends to the optimal value predicted by information theory.

The dictionary size is not bounded in this scheme. In practice, the dictionary is limited to a fixed size, like 4096 (so that each index can be encoded in 12 bits). Beyond that point, the transmissions continue in the same manner, but the dictionary is not updated. Also, in practical implementations, the dictionary is initially populated by all the symbols of the alphabet.

There are a number of variations of the Lempel-Ziv algorithm, all having the prefix LZ. What I have described here is known as LZ78 [51]. Many popular compression programs —Unix utility "compress", "gzip", Windows "Winzip"— are based on some variant of the Lempel-Ziv algorithm. Another algorithm, due to Burrows and Wheeler [9], is used in the popular "bzip" utility.

Implementation of the Dictionary We develop a data structure to implement the dictionary and the two operations on it: (1) from a given text find the (shortest) string that is not in the dictionary, and (2) add a new entry to the dictionary. The data structure is a special kind of tree (sometimes called a "trie"). Associated with each node of the tree is a word of the dictionary and its index; associated with each branch is a symbol, and branches from a node have different associated symbols. The root node has the word $\langle\rangle$ (empty string) and index 0 associated with it. The word associated with any node is the sequence of symbols on the path to that node. Initially, the tree has only the root node.

Given a text string, the sender starts matching its symbols against the symbols at the branches, starting at the root node. The process continues until a node, n, is reached from which there is no branch labelled with the next input symbol, s. At this point, index of n and the symbol s are transmitted. Additionally, node n is extended with a branch labelled s.

Consider the tree shown in Figure 1.3. If the text is *taccag#*, the prefix *tac* matches until the node with index 8. Therefore, index 8 and the next symbol, c, are transmitted. The tree is updated by adding a branch out of node 8, labelled c; the new node acquires the highest index, 9.

Exercise 5

1. Is it necessary to maintain the word at each node?
2. If your input alphabet is large, it will be non-trivial to look for a branch out of a node that is labeled with a specific symbol. Devise an efficient implementation of the tree in this case.
3. Suppose that the string *Ophelia* appears in the text, but none of its prefixes do. How many occurrences of this string should be seen before it is encoded as a word? □

Acknowledgment I am grateful to Thierry Joffrain for helping me write part of Section 1.2.

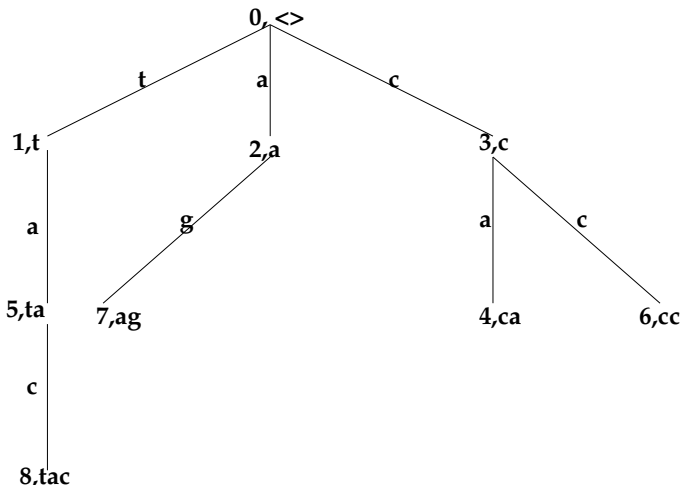


Figure 1.3: Implementation of the dictionary in Lempel-Ziv Algorithm

Chapter 2

Error Detection and Correction

2.1 Introduction

The following description from Economist, July 3rd, 2004, captures the essence of error correction and detection, the subject matter of this chapter. "On July 1st [2004], a spacecraft called *Cassini* went into orbit around Saturn —the first probe to visit the planet since 1981. While the rockets that got it there are surely impressive, just as impressive, and much neglected, is the communications technology that will allow it to transmit its pictures millions of kilometers back to Earth with antennae that use little more power than a light-bulb.

To perform this transmission through the noisy vacuum of space, *Cassini* employs what are known as error-correcting codes. These contain internal tricks that allow the receiver to determine whether what has been received is accurate and, ideally, to reconstruct the correct version if it is not."

First, we study the logical operator *exclusive-or*, which plays a central role in error detection and correction. The operator is written as $\oplus$ in these notes. It is a binary operator, and its truth table is shown in Table 2.1. Encoding *true* by 1 and *false* by 0, we get Table 2.2, which shows that the operator is addition modulo 2, i.e., addition in which you discard the carry.

In all cases, we apply $\oplus$ to bit strings of equal lengths, which we call *words*. The effect is to apply $\oplus$ to the corresponding bits independently. Thus,

	F	T
F	F	T
T	T	F

Table 2.1: Truth table of exclusive-or

	0	1
0	0	1
1	1	0

Table 2.2: Exclusive-or as addition modulo 2

$$\begin{array}{ll} & 0\ 1\ 1\ 0 \\ \oplus & \\ & 1\ 0\ 1\ 1 \\ = & \\ & 1\ 1\ 0\ 1 \end{array}$$

2.1.1 Properties of Exclusive-Or

In the following expressions x, y and z are words of the same length, 0 is a word of all zeros, and 1 is a word of all ones. $\overline{x}$ denotes the word obtained from x by complementing each of its bits.

- $\oplus$ is commutative: $x \oplus y = y \oplus x$
- $\oplus$ is associative: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$
- zero and complementation: $x \oplus 0 = x$, $x \oplus 1 = \overline{x}$
- inverse: $x \oplus x = 0$, $x \oplus \overline{x} = 1$
- distributivity over complementation: $\overline{(x \oplus y)} = (\overline{x} \oplus y)$
- Opposite of equivalence: $(x \oplus y) = \overline{(x \equiv y)}$

From the inverse property, we can regard $\oplus$ as subtraction modulo 2.

2.1.2 Dependent Set

A nonempty set of words, W, is *dependent* iff $\hat{W} = 0$, where $\hat{W}$ is the exclusive-or of all the words in W. Dependent sets are used in two applications later in these notes, in Sections 2.4 and 2.6.2.

Observation W is dependent iff for every partition of W into subsets X and Y, $\hat{X} = \hat{Y}$.
Proof: Let X and Y be any partition of W.

$$\begin{array}{ll} & \hat{W} = 0 \\ \equiv & \{X, Y \text{ is a partition of } W\text{; so } \hat{W} = \hat{X} \oplus \hat{Y}\} \\ & \hat{X} \oplus \hat{Y} = 0 \\ \equiv & \{\text{add } \hat{Y} \text{ to both sides of this equation}\} \end{array}$$

x	$=$	α	1	β
$x \oplus u$	$=$	α	0	γ
u	$=$	0s	1	$\beta \oplus \gamma$

Table 2.3: Computing $x \oplus x \oplus u$

$$\begin{array}{ll} & \hat{X} \oplus \hat{Y} \oplus \hat{Y} = \hat{Y} \\ \equiv & \{\hat{Y} \oplus \hat{Y} = 0 \text{ and } \hat{X} \oplus 0 = \hat{X}\} \\ & \hat{X} = \hat{Y} \end{array}$$

□

The proof of the following observation is similar to the one above and is omitted.

Observation W is dependent iff there is a partition of W into subsets X and Y, $\hat{X} = \hat{Y}$. □

Note: The two observations above say different things. The first one says that if W is dependent then for *all* partitions into X and Y we have $\hat{X} = \hat{Y}$, and, conversely, if for *all* partitions into X and Y we have $\hat{X} = \hat{Y}$, then W is dependent. The second observation implies a stronger result than the latter part of the first observation: if there exists any (not all) partition into U and V such that $\hat{U} = \hat{V}$, then W is dependent. □

Exercise 6

1. Show that $\overline{x} \oplus \overline{y} = x \oplus y$, and $(\overline{x} \equiv \overline{y}) = (x \equiv y)$.
2. Show that $(x \oplus y = x \oplus z) \equiv (y = z)$. As a corollary, prove that $(x \oplus y = 0) \equiv (x = y)$.
3. What is the condition on x and u so that $(x \oplus u) < x$, where x and u are numbers written in binary?
4. Let W' be a set obtained from a dependent set W by either removing an element or adding an element. Given W' determine W.

Solution to Part 3 of Exercise 1 Since $(x \oplus u) < x$, there is a bit position where x has a 1 and $x \oplus u$ has a 0, and all bits to the left of this bit are identical in x and $x \oplus u$. So, x is of the form $\alpha 1 \beta$ and $x \oplus u$ is of the form $\alpha 0 \gamma$. Then, taking their exclusive-or, see Table 2.3, we find that u has a string of zeros followed by a single 1 and then another string $(\beta \oplus \gamma)$. Comparing x and u in that table, x has a 1 in the position where the leading 1 bit of u appears. This is the only relevant condition. It is not necessary that x be larger than u; construct an example where $x < u$. □

Exercise 7

There are 100 men standing in a line, each with a hat on his head. Each hat is either *black* or *white*. A man can see the hats of all those in front of him, but not his own hat nor of those behind him. Each man is asked to guess the color of his hat, in turn from the back of the line to the front. He shouts his guess which every one can hear. Devise a strategy to maximize the number of correct guesses.

A possible strategy is as follows. Number the men starting at 0 from the back to the front. Let the guess of $2i$ be the color of $(2i+1)$'s hat, and $(2i+1)$'s guess is what he heard from $2i$. So, $(2i+1)$'s guess is always correct; thus, half the guesses are correct. We do considerably better in the solution, below.

Solution Number the men 0 through N from back to front. Let

h_i, 0 or 1, be the color of i's hat,
g_i, be i's guess,
H_i the exclusive-or of all succeeding hat colors, i.e., $(\oplus j : i < j \leq N : h_j)$
G_i the exclusive-or of all preceding guesses , i.e., $(\oplus j : 0 \leq j < i : g_j)$
Take $H_N = 0$ and $G_0 = 0$

Note that every i can compute G_i (from the guesses he has heard) and H_i (from the hats he can see). For every i, including $i = 0$ and $i = N$, the guess g_i is $G_i \oplus H_i$. We claim that g_i is a correct guess, i.e., $g_i = h_i$, for all i, $i > 0$.

First, observe that

$$\begin{array}{ll} & H_i = H_{i+1} \oplus h_{i+1} \\ \Rightarrow & \{\text{Append } h_{i+1} \text{ to both sides, and note that } h_{i+1} \oplus h_{i+1} = 0\} \\ & H_i \oplus h_{i+1} = H_{i+1} \end{array}$$

Now, we prove that $g_{i+1} = h_{i+1}$, for all i, $i \geq 0$.

$$\begin{array}{ll} & g_{i+1} \\ = & \{\text{definition of } g\} \\ & G_{i+1} \oplus H_{i+1} \\ = & \{\text{from the definition of } G,\ G_{i+1} = g_i \oplus G_i\} \\ & g_i \oplus G_i \oplus H_{i+1} \\ = & \{\text{from above, } H_{i+1} = H_i \oplus h_{i+1}\} \\ & g_i \oplus G_i \oplus H_i \oplus h_{i+1} \\ = & \{\text{from the definition of } g,\ g_i = G_i \oplus H_i.\ \text{So, } g_i \oplus G_i \oplus H_i = 0\} \\ & h_{i+1} \end{array}$$

2.2 Small Applications

2.2.1 Complementation

To complement some bit of a word is to flip it, from 1 to 0 or 0 to 1. To selectively complement the bits of x where y has a 1, simply do

$x := x \oplus y$

From symmetry of the right side, the resulting value of x is also a complementation of y by x. If y is a word of all 1s, then $x \oplus y$ is the complement of (all bits of) x; this is just an application of the law: $x \oplus 1 = \overline{x}$.

Suppose we want to construct a word w from x, y and u as follows. Wherever u has a 0 bit choose the corresponding bit of x, and wherever it has 1 choose from y, see the example below.

$u = 0\ 1\ 0\ 1$
$x = 1\ 1\ 0\ 0$
$y = 0\ 0\ 1\ 1$
$w = 1\ 0\ 0\ 1$

Then w is, simply, $((x \oplus y) \wedge u) \oplus x$, where $\wedge$ is applied bit-wise.

Exercise 8
Prove this result. □

2.2.2 Toggling

Consider a variable x that takes two possible values, m and n. We would like to *toggle* its value from time to time: if it is m , it becomes n and vice versa. There is a neat way to do it using exclusive-or. Define a variable t that is initially set to $m \oplus n$ and never changes.

toggle:: $x := x \oplus t$

To see why this works, check out the two cases: before the assignment, let the value of x be m in one case and n in the other. For $x = m$, the toggle sets x to $m \oplus t$, i.e., $m \oplus m \oplus n$, which is n. The other case is symmetric.

Exercise 9
Variable x assumes the values of p, q and r in cyclic order, starting with p. Write a code fragment to assign the next value to x, using $\oplus$ as the primary operator in your code. You will have to define additional variables and assign them values along with the assignment to x.

Solution Define two other variables y and z whose values are related to x's by the following invariant:

$x, y, z = t, t \oplus t', t \oplus t''$

where t' is the next value in cyclic order after t (so, $p' = q$, $q' = r$ and $r' = p$), and t'' is the value following t'. The invariant is established initially by letting

$x, y, z = p, p \oplus q, p \oplus r$

The cyclic assignment is implemented by

$x := x \oplus y;$
$y := y \oplus z;$
$z := y \oplus z$

Show that if $x, y, z = t, t \oplus t', t \oplus t''$ before these assignments, then $x, y, z = t', t' \oplus t'', t' \oplus t$ after the assignments (note: $t''' = t$). □

2.2.3 Exchange

Here is a truly surprising application of $\oplus$. If you wish to exchange the values of two variables you usually need a temporary variable to hold one of the values. You can exchange *without* using a temporary variable. The following assignments exchange the values of x and y.

$x := x \oplus y;$
$y := x \oplus y;$
$x := x \oplus y$

To see that this program actually exchanges the values, suppose the values of x and y are X and Y before the exchange. The following annotated program shows the values they have at each stage of the computation; I have used backward substitution to construct this annotation. The code is to the left and the annotation to the right in a line.

	$y = Y,\ (x \oplus y) \oplus y = X$, i.e., $x = X,\ y = Y$
$x := x \oplus y;$	
	$x \oplus (x \oplus y) = Y,\ (x \oplus y) = X$, i.e., $y = Y,\ (x \oplus y) = X$
$y := x \oplus y;$	
	$x \oplus y = Y,\ y = X$
$x := x \oplus y$	
	$x = Y,\ y = X$

2.2.4 Storage for Doubly-Linked Lists

Each node x in a doubly-linked list stores a data value, a left pointer, $x.left$, to a node and a right pointer, $x.right$, to a node. One or both pointers may be nil, a special value. A property of the doubly-linked list is that for any node x

$x.left \neq nil \ \Rightarrow \ x.left.right = x$
$x.right \neq nil \ \Rightarrow \ x.right.left = x$

Typically, each node needs storage for the data value and for two pointers. The storage for two pointers can be reduced to the storage needed for just one pointer; store $x.left \oplus x.right$ at x. How do we retrieve the two pointer values from this one value? During a computation, node x is reached from either the left or the right side; therefore, either $x.left$ or $x.right$ is known. Applying $\oplus$ to

the known pointer value and $x.left \oplus x.right$ yields the other pointer value; see the treatment of toggling in Section 2.2.2. Here, *nil* should be treated as 0.

We could have stored $x.left + x.right$ and subtracted the known value from this sum; exclusive-or is faster to apply and it avoids overflow problems.

Sometimes, nodes in a doubly-linked list are reached from some node outside the list; imagine an array each of whose entries points to a node in a doubly-linked list. The proposed pointer compression scheme is not useful then because you can reach a node without knowing the value of any of its pointers.

Note: These kinds of pointer manipulations are often prevented by the compiler of a high-level language through type checks. I don't advocate such manipulations except when you are programming in an assembly language, and you need to squeeze out the last drop of performance. Even then see if there are better alternatives; often a superior data structure or algorithm gives you far better performance than clever tricks![1] □

2.2.5 Hamming Distance

The *Hamming distance* —henceforth, simply called *distance*— between two words is the number of positions where they differ. Thus the distance between 1 0 0 1 and 1 1 0 0 is 2. This is the number of 1s in 1 0 0 1 $\oplus$ 1 1 0 0, which is 0 1 0 1.

Distance is a measure of how similar two words are; smaller the distance greater the similarity. Observe the following properties of distance. Below, x, y and z are words and $d(x, y)$ is the distance between x and y.

- $(d(x, y) = 0) \equiv (x = y)$
- $d(x, y) \geq 0$
- $d(x, y) = d(y, x)$
- (Triangle Inequality) $d(x, y) + d(y, z) \geq d(x, z)$

In order to prove these properties, you need some facts about the number of 1s in a word. Let $count(x)$ be the number of 1s in x. Then, $d(x, y) = count(x \oplus y)$. We have the following properties of *count*.

$count(x) \geq 0$
$(count(x) = 0) \equiv (x = 0)$
$count(x) + count(y) = count(x \oplus y) + 2f$, for some nonnegative f.

The first two properties are easy to see. For the last property, let f be the number of positions where x and y both have 1s. In $x \oplus y$, the bit values at

[1] "The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague". From "The Humble Programmer" by Edsger W. Dijkstra, 1972 Turing Award lecture [14].

these positions are 0, and at every other position the number of 1s in $x \oplus y$ is same as the number of 1s in that position of x and y combined. Therefore, $x \oplus y$ has $2f$ fewer ones than x and y combined.

It follows from the last property that

$count(x) + count(y) \geq count(x \oplus y)$
$count(x) + count(y)$ has the same parity (even or odd) as $count(x \oplus y)$

The proof of Triangle Inequality is now immediate.

$$
\begin{array}{ll}
 & count(x) + count(y) \geq count(x \oplus y) \\
\Rightarrow & \{\text{replace } x \text{ by } x \oplus y \text{ and } y \text{ by } y \oplus z\} \\
 & count(x \oplus y) + count(y \oplus z) \geq count(x \oplus y \oplus y \oplus z) \\
\equiv & \{\text{simplify}\} \\
 & count(x \oplus y) + count(y \oplus z) \geq count(x \oplus z) \\
\equiv & \{d(x,y) = count(x \oplus y); \text{ similarly for the other terms}\} \\
 & d(x,y) + d(y,z) \geq d(x,z)
\end{array}
$$

Hamming distance is essential to the study of error detection (Section 2.5) and error correction (Section 2.6).

Exercise 10

A word x has *even parity* if $count(x)$ is even, otherwise it has *odd parity*. Show that two words of identical parity (both even or both odd) have even distance, and words of different parity have odd distance.

Solution In the following proof we start with a property of *count*.

$$
\begin{array}{ll}
 & count(x) + count(y) \text{ has the same parity (even or odd) as } count(x \oplus y) \\
\Rightarrow & \{\text{writing } even(n) \text{ to denote that number } n \text{ is even}\} \\
 & even(count(x) + count(y)) \;\equiv\; even(count(x \oplus y)) \\
\equiv & \{\text{for any two integers } p \text{ and } q,\ even(p+q) \;=\; (even(p) \;\equiv\; even(q)); \\
 & \quad \text{let } p \text{ be } count(x) \text{ and } q \text{ be } count(y)\} \\
 & (even(count(x)) \;\equiv\; even(count(y))) \;\equiv\; even(count(x \oplus y)) \\
\equiv & \{count(x \oplus y) = d(x,y)\} \\
 & (even(count(x)) \;\equiv\; even(count(y))) \;\equiv\; even(d(x,y))
\end{array}
$$

The term $even(count(x))$ stands for "x has even parity". Therefore, the first term in the last line of the above proof, $(even(count(x)) \;\equiv\; even(count(y)))$, denotes that x and y have identical parity. Hence, the conclusion in the above proof says that the distance between x and y is even iff x and y have identical parity. □

Exercise 11

Let $w_1, w_2, \ldots, w_N$ be a set of *unknown* words. Let W_i be the exclusive-or of all the words except w_i, $1 \leq i \leq N$. Given $W_1, W_2, \ldots, W_N$, can you determine the values of $w_1, w_2, \ldots, w_N$? You can only apply $\oplus$ on the words. You may

prefer to attack the problem without reading the following hint.

Hint:
1. Show that the problem can be solved when N is even.
2. Show that the problem cannot be solved when N is odd.

A more general problem:
Investigate how to solve a general system of equations that use $\oplus$ as the only operator. For example, the equations may be:

$$\begin{array}{ll} w_1 \oplus w_2 \oplus w_4 & = 1\ 0\ 0\ 1\ 1 \\ w_1 \oplus w_3 & = 1\ 0\ 1\ 1\ 0 \\ w_2 \oplus w_3 & = 0\ 0\ 0\ 0\ 1 \\ w_3 \oplus w_4 & = 1\ 1\ 0\ 1\ 1 \end{array}$$

Solution Let S denote the exclusive-or of all the unknowns, i.e., $S = w_1 \oplus w_2 \oplus \ldots \oplus w_N$. Then $W_i = S \oplus w_i$.

1. For even N:

$$\begin{array}{ll} & W_1 \oplus W_2 \oplus \ldots \oplus W_N \\ = & \{W_i = S \oplus w_i\} \\ & (S \oplus W_1) \oplus (S \oplus W_2) \oplus \ldots \oplus (S \oplus W_N) \\ = & \{\text{Regrouping terms}\} \\ & (S \oplus S \oplus \ldots \oplus S)\ \oplus\ (w_1 \oplus w_2 \oplus \ldots \oplus w_N) \\ = & \{\text{the first operand has an even number of } S\} \\ & 0 \oplus (w_1 \oplus w_2 \oplus \ldots \oplus w_N) \\ = & \{\text{the last operand is } S\} \\ & S \end{array}$$

   Once S is determined, we can compute each w_i because

$$\begin{array}{ll} & S \oplus W_i \\ = & \{W_i = S \oplus w_i\} \\ & S \oplus S \oplus w_i \\ = & \{S \oplus S = 0\} \\ & w_i \end{array}$$

2. For odd N: We show that any term that we compute is exclusive-or of some subset of $w_1, w_2, \ldots, w_N$, and *the subset size is even.* Therefore, we will never compute a term that represents, say, w_1 because then the subset size is odd.

   To motivate the proof, suppose we have $N = 5$, so $W_1 = w_2 \oplus w_3 \oplus w_4 \oplus w_5$, $W_2 = w_1 \oplus w_3 \oplus w_4 \oplus w_5$, $W_3 = w_1 \oplus w_2 \oplus w_3 \oplus w_4$, $W_4 = w_1 \oplus w_2 \oplus w_3 \oplus w_5$, $W_5 = w_1 \oplus w_2 \oplus w_3 \oplus w_4$. Initially, each of the terms, W_1, W_2 etc., is represented by a subset of unknowns of size 4. Now, suppose we compute a new term, $W_1 \oplus W_4$; this represents $w_2 \oplus w_3 \oplus w_4 \oplus w_5 \oplus w_1 \oplus w_2 \oplus w_3 \oplus w_5$, which is same as $w_1 \oplus w_4$, again a subset of even number of terms.

The proof is as follows. Initially the proposition holds because each W_i is the exclusive-or of all but one of the unknowns, namely w_i; so the corresponding subset size is $N-1$, which is even since N is odd.

Whenever we apply $\oplus$ to any two terms: (1) either their subsets have no common unknowns, so the resulting subset contains all the unknowns from both subsets, and its size is the sum of both subset sizes, which is even, or (2) the subsets have some number of common unknowns, which get cancelled out from both subsets, again yielding an even number of unknowns for the resulting subset. □

2.2.6 The Game of Nim

The game of Nim is a beautiful illustration of the power of the exclusive-or operator.

The game is played by two players who take turns in making moves. Initially, there are several piles of chips and in a move a player may remove any positive number of chips from a single pile. A player loses when he can't make a move, i.e., all piles are empty. We develop the conditions for a specific player to win.

Suppose there is a single pile. The first player wins by removing all chips from that pile. Now suppose there are two piles, each with one chip, call this initial state (1,1). The first player is forced to empty out one pile, and the second player then removes the chip from the other pile, thus winning the game. Finally, consider two piles, one with one chip and the other with two chips. If the first player removes all chips from either pile, he loses. But if he removes one chip from the bigger pile, he creates the state (1,1) which leads to a defeat for the second player, from the previous argument.

The Underlying Mathematics Consider the number of chips in a pile as a word (a bit string) and take the exclusive-or of all the words. Call the state *losing* if the result is 0, *winning* otherwise. Thus, the state (1,1) results in 0, a losing state, whereas (1,2) gives $0\ 1 \oplus 1\ 0 = 1\ 1$, which is a winning state. The final state, where all piles are empty, is a losing state. The mnemonics, *losing* and *winning*, signify the position of a player: a player who has to make a move in a winning state has a winning strategy, i.e., if he makes the right moves he wins no matter what his opponent does; a player in a losing state will definitely lose provided his opponent makes the right moves. So, one of the players has a winning strategy based on the initial state. Of course, either player is allowed to play stupidly and squander a winning position.

The proof of this result is based on the following state diagram. We show that any possible move in a losing state can only lead to a winning state, thus a player who has to move in this state cannot do anything but hope that his opponent makes a mistake! A player in a winning state has at least one move to transform the state to losing; of course, he can make a wrong move and remain in the winning state, thus handing his opponent the mistake he was hoping for. Next, we prove the claims made in this diagram.

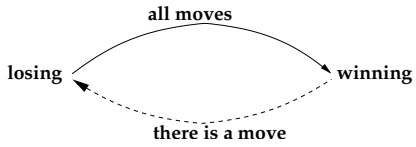


Figure 2.1: State transitions in the game of Nim

A move reduces a pile of p chips to q chips, $0 \leq q < p$. Let the exclusive-or of the remaining piles be s. Before the move, exclusive-or of all piles was $s \oplus p$. After the move it is $s \oplus q$. First, we show that in a losing state, i.e., $s \oplus p = 0$, all possible moves establish a winning state, i.e., $s \oplus q \neq 0$.

$$\begin{array}{ll} & s \oplus q \\ = & \{p \oplus p = 0\} \\ & s \oplus q \oplus p \oplus p \\ = & \{s \oplus p = 0\} \\ & p \oplus q \\ \neq & \{p \neq q\} \\ & 0 \end{array}$$

Now, we show that there is a move in the winning state to take it to losing state. Let the exclusive-or of all piles be u, $u \neq 0$. Let x be any pile that has a 1 in the same position as the leading 1 bit of u. So,

$$\begin{array}{rcl} u & = & 0's\ 1\ \ \gamma \\ x & = & \alpha\ \ \ 1\ \ \beta \end{array}$$

The winning move is to replace x by $x \oplus u$. We show that (1) $x \oplus u < x$, and (2) the exclusive-or of the resulting set of piles is 0, i.e., the state after the move is a losing state.

Proof of (1): $x \oplus u = \alpha\ 0(\beta \oplus \gamma)$. Comparing x and $x \oplus u$, we have $x \oplus u < x$.

Proof of (2): The exclusive-or of the piles before the move is u; so, the exclusive-or of the piles except x is $x \oplus u$. Hence, the exclusive-or of the piles after the move is $(x \oplus u) \oplus (x \oplus u)$, which is 0.

Exercise 12
In a winning state let y be a pile that has a 0 in the same position as the leading bit of u. Show that removing any number of chips from y leaves a winning state.

Solution The forms of u and y are as follows.

$$\begin{array}{rcl} u & = & 0s\ \ 1\ \ \gamma \\ y & = & \alpha\ \ \ 0\ \ \beta \end{array}$$

Suppose y is reduced to y' and the exclusive-or of the resulting set is 0. Then $u \oplus y \oplus y' = 0$, or $y' = u \oplus y$. Hence, $y' = \alpha\ 1\ (\beta \oplus \gamma)$. So, $y' > y$; that is , such a move is impossible. □

2.3 Secure Communication

The problem in secure communication is for a sender to send a message to a receiver so that no eavesdropper can read the message during transmission. It is impossible to ensure that no one else can see the transmission; therefore, the transmitted message is usually encrypted so that the eavesdropper cannot decipher the real message. In most cases, the sender and the receiver agree on a transmission protocol; the sender encrypts the message in such a fashion that only the receiver can decrypt it.

In this section, I describe a very simple encryption (and decryption) scheme whose only virtue is simplicity. Usually, this form of transmission can be *broken* by a determined adversary. There are now very good methods for secure transmission, see Rivest, Shamir and Adelman [41].

The sender first converts the message to be sent to a bit string, by replacing each symbol of the alphabet by its ascii representation, for instance. This string is usually called the *plaintext*. Next, the plaintext is broken up into fixed size blocks, typically around 64 bits in length, which are then encrypted and sent. For encryption, the sender and the receiver agree on a key k, which is a bit string of the same length as the block. To send a string x, the sender transmits y, where $y = x \oplus k$. The receiver, on receiving y, computes $y \oplus k$ which is $(x \oplus k) \oplus k$, i.e., x, the original message. An eavesdropper can only see y which appears as pure gibberish. The transmission can only be decrypted by some one in possession of key k.

There are many variations on this simple scheme. It is better to have a long key, much longer than the block length, so that successive blocks are encrypted using different strings. When the bits from k run out, wrap around and start reusing the bits of k from the beginning. Using a longer key reduces the possibility of the code being broken.

This communication scheme is simple to program; in fact, encryption and decryption have the same program. Each operation is fast, requiring time proportional to a block length for encryption (and decryption). Yet, the scheme has significant drawbacks. Any party who has the key can decode the message. More important, any one who can decode a single block can decode all blocks (assuming that the key length is same as the block length), because given x and y where $y = x \oplus k$, k is simply $x \oplus y$. Also, the sender and the receiver will have to agree on a key before the transmission takes place, so the keys have to be transmitted first in a secure manner, a problem known as *key exchange*. For these and other reasons, this form of encryption is rarely used in high security applications.

Exercise 13

The following technique has been suggested for improving the security of transmission. The sender encrypts the first block using the key k. He encrypts subsequent blocks by using the previous encrypted block as the key. Is this secure? How about using the plaintext of the previous block as the key? Suppose a single block is deciphered by the eavesdropper; can he then decipher all

blocks, or all subsequent blocks? □

2.4 RAID Architecture

The following scenario is common in corporate data centers. A large database, consisting of millions of records, is stored on a number of disks. Since disks may fail, data is stored on backup disks also. One common strategy is to partition the records of the database and store each partition on a disk, and also on a backup disk. Then, failure of one disk causes no difficulty. Even when multiple disks fail, the data can be recovered provided both disks for a partition do not fail.

There is a different strategy, known as RAID, that has gained popularity because it needs only one additional disk beyond the primary data disks, and it can tolerate failure of any one disk.

Imagine that the database is a matrix of bits, where each row represents a record, and each column a specific bit in all records. Store each column on a separate disk and store the exclusive-or of all columns on a backup disk. Let c_i denote the ith column, $1 \leq i \leq N$, in the database. Then the backup column, c_0 is given by $c_0 = c_1 \oplus \ldots \oplus c_N$. Therefore, the set of columns, $c_0 \ldots c_N$, is a dependent set, see Section 2.1.2. Then, any column c_i, $0 \leq i \leq N$, is the exclusive-or of the remaining columns. Therefore, the contents of any failed disk can be reconstructed from the remaining disks.

2.5 Error Detection

Message transmission is vulnerable to noise, which may cause portions of a message to be altered. For example, message 1 1 0 0 1 may become 1 0 1 0 1. In this section, we study methods by which a receiver can determine that the message has been altered, and thus request retransmission. In the next section, we discuss methods by which a receiver can correct (some of) the errors, thus avoiding retransmission.

Typically, a long message is broken up into fixed size blocks (extra bits, which can be distinguished from the real ones, are added at the end of the message so that the string fits exactly into some number of blocks; typically, the extra bits carry information about the length and composition of the block). Henceforth, each block is transmitted independently, and we concentrate on the transmission of a single block.

2.5.1 Parity Check Code

Consider the following input string where spaces separate the blocks.

011 100 010 111

The sender appends a bit at the end of each block so that each 4-bit block has an even number of 1s. This additional bit is called a *parity* bit, and each

block is said to have *even parity*. After addition of parity bits, the input string shown above becomes,

0110 1001 0101 1111

This string is transmitted. Suppose two bits are flipped during transmission, as shown below; the flipped bits are underlined.

0110 100$\underline{0}$ 0101 $\underline{0}$111

Note that the flipped bit could be a parity bit or one of the original ones. Now each erroneous block has odd parity, and the receiver can identify all such blocks. It then asks for retransmissions of those blocks.

If two bits (or any even number) of a block get flipped, the receiver cannot detect the error. This is a serious problem, so simple parity check is rarely used. In practice, the blocks are much longer (than 3, shown here) and many additional bits are used for error detection.

Is parity coding any good? How much is the error probability reduced if you add a single parity bit? The analysis here uses elementary probability theory. Let p be the probability of error in the transmission of a single bit[2]. The probability of correct transmission of a single bit is q, where $q = 1 - p$. The probability of correct transmission of a b bit block is q^b. Therefore, the probability that there is an undetected error in the block is $1 - q^b$. For $p = 10^{-4}$ and $b = 12$, this probability is around 1.2×10^{-3}.

With the addition of a parity bit, we have to send b+1 bits. But we can detect one error if it arises. So, the probability of undetected error is 1− the probability that there is no error in transmission − the probability that there is a single error in transmission. The probability that there is no error in transmission is q^{b+1}. The probability that there is a single error can be computed as follows: it is the probability that the first bit is incorrectly transmitted and the remaining bits are correctly transmitted ($p \times q^b$) plus the probability that the second bit is incorrectly transmitted and the remaining bits are correctly transmitted (same probability) plus .... Therefore, the probability of a single error is $(b+1) \times p \times q^b$. Hence, the probability of undetected error is $1 - q^{b+1} - (b+1) \times p \times q^b$. Setting $b, p, q = 12, 10^{-4}, 1 - 10^{-4}$, this probability is around 9.4×10^{-7}, several orders of magnitude smaller than 1.2×10^{-3}.

2.5.2 Horizontal and Vertical Parity Check

A simple generalization of the simple parity check scheme is described next. We regard the data as a matrix of bits, not just a linear string. For instance, we may break up a 16 bit block into 4 subblocks, each of length 4. We regard each subblock as the row of a matrix, so, column i is the sequence of ith bits from each subblock. Then we add parity bits to each row and column, and a

[2]I am assuming that all errors are independent, a thoroughly false assumption when burst errors can arise.

1	0	1	1	1
0	1	1	1	1
1	1	1	0	1
0	0	1	1	0
0	0	0	1	1

Table 2.4: Adding parity bits to rows and columns

single bit for the entire matrix. In Table 2.4, 4 subblocks of length 4 each are transformed into 5 subblocks of length 5 each.

We can now detect odd number of errors in rows *or* columns . If two adjacent bits in a row get altered, the row parity remains the same but the column parities for the affected columns are altered.

The most common use of this scheme is in transmitting a sequence of ascii characters. Each character is a 8-bit string, which we regard as a row. And 8 characters make up a block.

Exercise 14

Show an error pattern in Table 2.4 that will not be detected by this method. □

Exercise 15

Develop a RAID architecture based on two-dimensional parity bits. □

2.6 Error Correction

In many practical situations, retransmission is expensive or impossible. For example, when the sender is a spacecraft from a distant planet, the time of transmission can be measured in days; so, retransmission adds significant delay, and the spacecraft will have to store a huge amount of data awaiting any retransmission request. Even more impractical is to request retransmission of the music on a CD whose artist is dead.

2.6.1 A Naive Error-Correcting Code

When retransmission is not feasible, the sender encodes the messages in such a way that the receiver can detect and correct some of the errors. As an example, suppose that the sender plans to send a 2-bit message. Adding a parity bit increases the block length to 3. Repeating the original 2-bit message after that gives a 5-bit block, as shown in Table 2.5.

Each of the possible blocks —in this case, 5-bit blocks— is called a *codeword*. Codewords are the only possible messages (blocks) that will be sent. So, if the sender plans to send 11, he will send 11011. In the example of Table 2.5, there are only four 5-bit codewords, instead of 32 possible ones. This means that it

Original	With Parity	Additional Bits
00	00**0**	00**0**00
01	01**1**	01**1**01
10	10**1**	10**1**10
11	11**0**	11**0**11

Table 2.5: Coding for error correction; parity bits are in bold

Codeword	Received Word	Hamming Distance
00000	11010	3
01101	11010	4
10110	11010	2
11011	11010	1

Table 2.6: Computing Hamming distance to codewords

will take longer to transmit a message, because many redundant bits will be transmitted. The redundancy allows us to detect and correct errors.

For the given example, we can detect two errors and correct one error in transmission. Suppose 11011 is changed to 11010. The receiver observes that this is not a codeword, so he has detected an error. He corrects the error by looking for the *nearest* codeword, the one that has the smallest Hamming distance from the received word. The computation is shown in Table 2.6. As shown there, the receiver concludes that the original transmission is 11011.

Now suppose two bits of the original transmission are altered, so that 11011 is changed to 10010. The computation is shown in Table 2.7. The receiver will detect that there is an error, but based on distances, he will assume that 10110 was sent. We can show that this particular encoding can correct one error only. The number of errors that can be detected/corrected depends on the Hamming distance among the codewords, as given by the following theorem.

Theorem 1 Let h be the Hamming distance between the nearest two codewords. It is possible to detect any number of errors less than h and correct any number of errors less than $h/2$.

Codeword	Received Word	Hamming Distance
00000	10010	2
01101	10010	5
10110	10010	1
11011	10010	2

Table 2.7: Hamming distance when there are two errors

Proof: The statement of the theorem is as follows. Suppose codeword x is transmitted and string y received.

1. if $d(x, y) < h$: the receiver can detect if errors have been introduced during transmission.

2. if $d(x, y) < h/2$: the receiver can correct the errors, if any. It picks the closest codeword to y, and that is x.

Proof of (1): The distance between any two distinct codewords is at least h. The distance between x and y is less than h. So, either $x = y$ or y is not a codeword. Therefore, the receiver can detect errors as follows: if y is a codeword, there is no error in transmission, and if y is not a codeword, the transmission is erroneous.

Proof of (2): We show that the closest codeword to y is x, i.e., for any other codeword z, $d(y, z) > d(x, y)$.

$$\begin{array}{ll} & d(x,y) + d(y,z) \\ \geq & \{\text{triangle inequality}\} \\ & d(x,z) \\ \geq & \{x \text{ and } z \text{ are codewords, and their distance is at least } h\} \\ & h \\ > & \{d(x,y) < h/2. \text{ Hence, } h > 2d(x,y)\} \\ & 2d(x,y) \end{array}$$

From this proof

$d(x, y) + d(y, z) > 2d(x, y)$, or
$d(y, z) > d(x, y)$

Exercise 16
Compute the nearest distance among the codewords in Table 2.5. □

It is clear from Theorem 1 that we should choose codewords to maximize h. But with a fixed block length, the number of codewords decreases drastically with increasing h. Table 2.8 shows the number of codewords for certain values of the block length and h; an entry like 72-79 denotes that the exact value is not known, but it lies within the given interval. Note that the decrease along a row is quite dramatic.

The coding scheme described in this section can correct one error, as follows. Suppose the sender wishes to send a bit string b; then it sends bpb, where p is the parity bit. At most one error is introduced during transmission, so the receiver sees $b'p'b''$ where the first b could be altered to b', p altered to p' and the second copy of b to b''. The receiver can deduce the following. If both $b'p'$ and $p'b''$ have even parity, then there is no error and $b' = b'' = b$. If one side, say $b'p'$ has odd parity and $p'b''$ has even parity then $b'' = b$; the error occurs in b'. If both $b'p'$ and $p'b''$ have odd parity, then the parity bit has been corrupted and $b' = b'' = b$.

Block length	$h = 3$	$h = 5$	$h = 7$
5	4	2	-
7	16	2	2
10	72-79	12	2
16	2560-3276	256-340	36-37

Table 2.8: Number of codewords for given block lengths and h

This scheme is extremely wasteful in the use of transmitted bits; only one error can be corrected at the expense of transmitting more than double the number of bits. We will see a more efficient scheme in the next section.

Exercise 17

Prove that the parity check code of Section 2.5.1 can be used to detect at most one error, but cannot be used to correct any error. □

2.6.2 Hamming Code

The coding scheme described in this section was developed by Hamming, a pioneer in Coding theory who introduced the notion of Hamming distance. It requires only $\log n$ extra bits, called *check bits*, and it corrects at most one error in an n-bit transmission. The novel idea is to transmit in the check bits the *positions* where the data bits are 1. Since it is impractical to actually transmit all the positions, we will instead transmit an encoding of them, using exclusive-or. Also, since the check bits can be corrupted as easily as the data bits, we treat them symmetrically, and also send the positions where the check bits are 1. More precisely, we regard each position number in the transmitted string as a word, and encode the check bits in such a way that the following rule is obeyed:

- HC Rule: the set of position numbers where the data bits and check bits are 1 form a dependent set, i.e., the exclusive-or of these positions, regarded as words, is 0 (see Section 2.1.2).

Let us look at an example where the HC rule has been applied.

Example Consider transmission of a 13-bit string, as shown in Table 2.9. The data bits are labeled d and the check bits c. The positions where 1s appear are labeled by *. They form a dependent set; check that

$$\begin{array}{ll} & 1\ 0\ 1\ 1 \quad (=11) \\ \oplus & \\ & 1\ 0\ 1\ 0 \quad (=10) \\ \oplus & \\ & 1\ 0\ 0\ 1 \quad (=9) \\ \oplus & \end{array}$$

0	0	1	1	1	1	0	1	0	1	1	0	1
d	d	d	d	d	c	d	d	d	c	d	c	c
13	12	11	10	9	8	7	6	5	4	3	2	1
		*	*	*	*		*		*	*		*

Table 2.9: Hamming code transmission

$$
\begin{array}{ll}
 & 1\,0\,0\,0 \quad (=8) \\
\oplus & \\
 & 0\,1\,1\,0 \quad (=6) \\
\oplus & \\
 & 0\,1\,0\,0 \quad (=4) \\
\oplus & \\
 & 0\,0\,1\,1 \quad (=3) \\
\oplus & \\
 & 0\,0\,0\,1 \quad (=1) \\
= & 0\,0\,0\,0
\end{array}
$$

□

The question for the sender is where to store the check bits (we have stored them in positions 8, 4, 2 and 1 in the example above) and how to assign values to them so that the set of positions is dependent. The question for the receiver is how to decode the received string and correct a possible error.

Receiver Let P be the set of positions where the transmitted string has 1s and P' where the received string has 1s. From the assumption that there is at most one error, we have either $P = P'$, $P' = P \cup \{t\}$, or $P = P' \cup \{t\}$, for some position t; the latter two cases arise when the bit at position t is flipped from 0 to 1, and 1 to 0, respectively. From rule HC, $\hat{P} = 0$, where $\hat{P}$ is the exclusive-or of the words in P.

The receiver computes $\hat{P'}$. If $P = P'$, he gets $\hat{P'} = \hat{P} = 0$. If $P' = P \cup \{t\}$, he gets $\hat{P'} = \hat{P} \oplus \{t\} = 0 \oplus \{t\} = t$. If $P = P' \cup \{t\}$, he gets $\hat{P} = \hat{P'} \oplus \{t\}$, or $\hat{P'} = \hat{P} \oplus \{t\} = 0 \oplus \{t\} = t$. Thus, in both cases where the bit at t has been flipped, $\hat{P'} = t$. If $t \neq 0$, the receiver can distinguish error-free transmission from erroneous transmission and correct the error in the latter case.

Sender We have seen from the previous paragraph that there should not be a position numbered 0, because then error-free transmission cannot be distinguished from one where the bit at position 0 has been flipped. Therefore, the positions in the transmitted string are numbered starting at 1. Each position is an n-bit word. And, we will employ n check bits.

Check bits are put at every position that is a power of 2 and the remaining bits are data bits. In the example given earlier, check bits are put at positions 1, 2, 4 and 8, and the remaining nine bits are data bits. So the position of any check bit as a word has a single 1 in it. Further, no two check bit position numbers have 1s in the same place.

Let C be the set of positions where the check bits are 1s and D the positions where the data bits are 1s. We know D, but we don't know C yet, because check bits have not been assigned values. We show next that C is uniquely determined from rule HC.

From rule HC, $\hat{C} \oplus \hat{D} = 0$. Therefore, $\hat{C} = \hat{D}$. Since we know D, we can compute $\hat{D}$. For the example considered earlier, $\hat{D} = 1101$. Therefore, we have to set the check bits so that $\hat{C} = 1101$. This is done by simply assigning the bit string $\hat{C}$ to the check bits in order from higher to lower positions; for the example, assign 1 to the check bit at positions 8, 4 and 1, and 0 to the check bit at position 2. The reason this rule works is that assigning a value v to the check bit at position 2^i, $i \geq 0$, in the transmitted string has the effect of setting the ith bit of $\hat{C}$ to v.

How many check bits do we need to transmit a given number of data bits? Let d be the number of data bits and c the number of check bits. With c check bits, we can encode 2^c positions, i.e., 0 through $2^c - 1$. Since we have decided not to have a position numbered 0 (see the discussion at the end of the "Receiver" and the beginning of the "Sender" paragraphs), the number of positions is at most $2^c - 1$. We have, $d + c \leq 2^c - 1$. Therefore, the number of data bits is no more than $2^c - 1 - c$.

2.6.3 Reed-Muller Code

You have probably emailed photographs or sent faxes. Such transmissions are always digital; text, image, audio, video are all converted first to bit strings and then transmitted. The receiver converts the received string to its original form. For text strings, conversion to and from bit strings is straightforward. For a still image, like a photograph or scanned document, the image is regarded as a matrix: a photograph, for instance may be broken up into 200 rows, each a strip, and each row may again be broken up into columns. It is not unusual to have over a million elements in a matrix for a photograph the size of a page. Each matrix element is called a *pixel* (for picture element). Each pixel is then converted to a bit string and the entire matrix is transmitted in either row-major or column-major order.

The conversion of a pixel into a bit string is not entirely straightforward; in fact, that is the subject matter of this section. In the most basic scheme, each pixel in a black and white photograph is regarded as either all black or all white, and coded by a single bit. This representation is acceptable if there are a large number of pixels, i.e., the resolution is fine, so that the eye cannot detect minute variations in shade within a pixel. If the resolution is low, say, an image of the size of a page is broken up into a 80×110 matrix, each pixel occupies around .01 square inch; the image will appear grainy after being converted at the receiver.

The Mariner 4 spacecraft, in 1965, sent 22 photographs of Mars, each one represented by a 200×200 matrix of pixels. Each pixel encoded 64 possible levels of brightness, and was transmitted as a 6-bit string. A single picture, consisting of $200 \times 200 \times 6$ bits, was transmitted at the rate of slightly over 8

$$H_1 = \left[\begin{array}{c|c} 1 & 1 \\ \hline 1 & 0 \end{array}\right]$$

Table 2.10: Hadamard matrix H_1

bits per second, thus requiring around 8 hours for transmission. The subsequent Mariners, 6, 7 and 9, did a much better job. Each picture was broken down to 700×832 pixels (i.e., 582,400 pixels per picture vs. 40,000 of Mariner 4) and each pixel of 6 bits was encoded by 32 bits, i.e., 26 redundant bits were employed for error detection and correction. The transmission rate was 16,200 bits per second. This takes around 18 minutes of transmission time per picture of much higher quality, compared to the earlier 8 hours.

Our interest in this section is in transmitting a single pixel so that any error in transmission can be detected and/or corrected. The emphasis is on correction, because retransmission is not a desirable option in this application. We study the simple Reed-Muller code employed by the later Mariners.

To motivate the discussion let us consider how to encode a pixel that has 8 possible values. We need only 3 bits, but we will encode using 8 bits, so as to permit error correction. As pointed out in Section 2.6.1, error correcting capability depends on the Hamming distance between the codewords. The 8-bit code we employ has distance 4 between *every* pair of codewords; so, we can detect 3 errors and correct 1. Error correction capability is low with 8-bit codewords. The Mariners employed 32-bit codewords, where the inter-word distance is 16; so, 15 errors could be detected and 7 corrected.

The codewords for the 8-bit Reed-Muller code are shown as rows of the matrix in Table 2.12. The rest of this section is devoted to the construction of 2^n codewords, $n \geq 1$, where the Hamming distance between any two codewords is exactly 2^{n-1}.

Hadamard Matrix

We will define a family of 0, 1 matrices H, where H_n is a $2^n \times 2^n$ matrix, $n \geq 0$. In the Reed-Muller code, we take each row of the matrix to be a codeword. We showed H_3 in Table 2.12.

The family H is defined recursively.

$$H_0 = \left[\ 1\ \right]$$

$$H_{n+1} = \left[\begin{array}{cc} H_n & H_n \\ H_n & \overline{H_n} \end{array}\right]$$

where $\overline{H_n}$ is the bit-wise complementation of H_n. Matrices H_1 and H_2 are shown in Tables 2.10 and 2.11, and H_3 appears in Table 2.12.

Hadamard matrices have many pleasing properties. The two that are of interest to us are: (1) H_n is symmetric for all n, and (2) the Hamming distance

$$H_2 = \left[\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ \hline 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{array}\right]$$

Table 2.11: Hadamard matrix H_2

$$H_3 = \left[\begin{array}{cccc|cccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ \hline 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{array}\right]$$

Table 2.12: Hadamard matrix H_3: 8-bit simple Reed-Muller code

between any two distinct rows in H_n, $n \geq 1$, is 2^{n-1}. Since the matrices have been defined recursively, it is no surprise that the proofs employ induction. I will leave the proof of (1) to you. Let us prove (2), using induction on n.

For $n = 1$: The result can be verified for H_1, shown in Table 2.10. The distance between the two rows is 2^{1-1}.

Before we prove the inductive case, we state a simple result. In the following, $d(x, y)$ stands for the Hamming distance between x and y.

Observation Let p and q be words of even length, say $2 \times t$. Then

$$\begin{aligned} d(p,q) = t &\equiv d(\overline{p},q) = t \\ d(p,q) = t &\equiv d(p,\overline{q}) = t \\ d(p,q) = t &\equiv d(\overline{p},\overline{q}) = t \end{aligned}$$ □

For $n + 1$, $n \geq 1$: Consider any two distinct rows x and y of H_{n+1}. Let x_l and x_r denote the left and right halves of row x (similarly, y_l and y_r are defined); see Figure 2.2. Then, $d(x, y) = d(x_l, y_l) + d(x_r, y_r)$. Now, both x_l and y_l are rows of H_n, because the left half of H_{n+1} has only such rows. But, x_r and y_r may be rows of H_n, $\overline{H_n}$ or both. We consider two cases:

(1) x_l and y_l are the same row of H_n: (See Case 2 in Figure 2.2.) Then $d(x_l, y_l) = 0$. Also $x_r = \overline{y_r}$. So, $d(x_r, y_r) = 2^n$. Hence, $d(x, y) = 2^n$. (Note: the inductive hypothesis has not been used in this proof.)

(2) x_l and y_l are distinct rows of H_n: Using the inductive hypothesis, $d(x_l, y_l) = 2^{n-1}$. The computation of $d(x_r, y_r)$ requires a case analysis. Either, (a) $x_r \in H_n$, $y_r \in H_n$, (b) $x_r \in H_n$, $y_r \in \overline{H_n}$, (c) $x_r \in \overline{H_n}$, $y_r \in H_n$, or (d) $x_r \in \overline{H_n}$, $y_r \in \overline{H_n}$. Therefore, either x_r or $\overline{x_r}$ is from H_n and so is y_r or $\overline{y_r}$.

In all cases, they correspond to different rows of H_n. By the inductive hypothesis, the two distinct rows of H_n they correspond to have distance 2^{n-1}. Using the observation above, $d(x_r, y_r) = 2^{n-1}$. So, $d(x, y) = d(x_l, y_l) + d(x_r, y_r) = 2^{n-1} + 2^{n-1} = 2^n$.

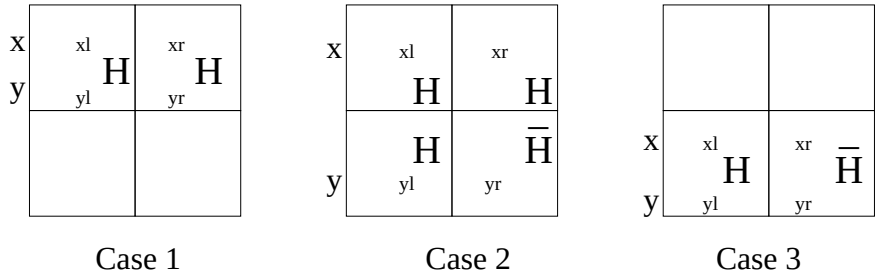


Figure 2.2: Proof of the distance property in Hadamard matrix

Chapter 3

Cryptography

3.1 Introduction

A central problem in secure communication is the following: *how can two parties, a sender and a receiver, communicate so that no eavesdropper can deduce the meaning of the communication*?

Suppose Alice has a message to send to Bob. Henceforth, we take a message to be a string over a specified alphabet; the string to be transmitted is usually called *plaintext*. The plaintext need not be a meaningful sentence. For instance, it could be a password or a credit card number. Any message could be intercepted by an eavesdropper, named Eve; so, it is not advisable to send the message in its plaintext form. Alice will encrypt the plaintext to create a *ciphertext*. Alice and Bob agree on a protocol, so that only Bob knows how to decrypt, i.e., convert the ciphertext to plaintext.

The goal of encryption and decryption is to make it hard (or impossible) for Eve to decrypt the ciphertext while making it easy for Alice to encrypt and Bob to decrypt. This means that Bob has some additional information, called a *key*, which Eve does not possess. When Alice and Bob share knowledge of the key, they are using a symmetric key system. Modern public key cryptography is asymmetric; encrypting uses a public key that is known to every one while decrypting requires a private key known only to the receiver.

The communication medium is really not important. Alice could write her message on a piece of paper (or on a clay tablet) and mail it physically; or she could send the message by email. Alice and Bob could engage in a telephone conversation in which only Alice speaks. Any communication medium is vulnerable, so security is achieved by choosing the encryption (and decryption) algorithms carefully.

a	c	d	k	n	t	w
d	t	k	w	n	a	c

Table 3.1: Substitution code for a subset of the Roman alphabet

3.2 Early Encryption Schemes

Secure communication has been important for at least 2,500 years, for military and romantic matters. In the very early days, the messenger simply hid the message, sometimes using invisible ink. As late as in the second world war, the Germans would often shrink a page of plaintext to a very small dot, less than 1mm in diameter, using photographic techniques, and then hide the dot within a full stop in a regular letter.

The aim of cryptography is not to hide the message but its meaning. Some of the earliest efforts simply scrambled the letters of the plaintext; this is called a transposition cypher. Thus, *attack at dawn* may be scrambled to *kntadatacwat* with the spaces removed. Since there are $n!$ permutations of n symbols it may seem impossible for even a computer to do a brute-force search to decode such messages. Bob, the intended recipient, can decrypt only if he is given the scrambling sequence. Transposition cyphers are, actually, quite easy to break; so they are rarely used except by schoolchildren.

3.2.1 Substitution Cyphers

A substitution cypher replaces a symbol (or a group of symbols) by another symbol (or a group of symbols). In the simplest case, each symbol is paired with another, and each occurrence of a symbol is replaced by its partner. Given the substitution code in Table 3.1, *attack at dawn* becomes *daadtw da kdcn*.

Julius Caesar used a very simple form of substitution in communicating with his generals. He replaced the ith symbol of the alphabet by symbol $(i+3)$ mod n, where n, the size of the alphabet, is 26. In general, of course, we can use any permutation of the alphabet, not merely a shift, as Caesar did. Caesar shift cypher is very easy to break; simply try all possible shifts. A general permutation is harder to crack, but not much harder as we will see. In all cases, the receiver must know the permutation in order to decrypt the message.

Cryptanalysis is the name given to unscrambling an intercepted message. For a substitution cypher, the eavesdropper can attempt a cryptanalysis based on the frequencies of letters in long plaintexts. Table 3.2 gives the frequencies (probability of occurrence) of letters in a piece of English text; clearly, different texts exhibit different frequencies, but the numbers given in the table are typical.

As can be seen in Table 3.2, e is the most common letter, followed by t and a. The cryptanalysis strategy is to replace the most common letter in the ciphertext by e, to see if it makes any sense. If not, then we try the remaining letters in sequence. For the plaintext *attack at dawn*, which has been converted to *daadtw da kdcn*, we first identify the most common letter in the ciphertext,

Letter	Frequency	Letter	Frequency	Letter	Frequency	Letter	Frequency
a	0.08167	b	0.01492	c	0.02782	d	0.04253
e	0.12702	f	0.02228	g	0.02015	h	0.06094
i	0.06966	j	0.00153	k	0.00772	l	0.04025
m	0.02406	n	0.06749	o	0.07507	p	0.01929
q	0.00095	r	0.05987	s	0.06327	t	0.09056
u	0.02758	v	0.00978	w	0.02360	x	0.00150
y	0.01974	z	0.00074				

Table 3.2: Frequencies of letters in English text

	a	c	d	k	n	t	w
d	3	1		1		1	
a	1		3				

Table 3.3: Frequencies of pairs of letters

d. Replacing each occurrence of d by e, the most common symbol, gives us the following string (I have used uppercase letters to show the guesses):

EaaEtw Ea kEcn

Since the second word is a two letter word beginning with E, which is uncommon except for proper names, we decide to abandon the guess that d is E. We try replacing d by t, the next most common symbol, to get

TaaTtw Ta kTcn

Now, it is natural to assume that a is really o, from the word *Ta*. This gives:

TOOTtw TO kTcn

It is unlikely that we can make any progress with the first word. So, we start fresh, with d set to the next most likely letter, a.

A variation of this scheme is to consider the frequencies of pairs of letters in the ciphertext, in the hope of eliminating certain possibilities. In Table 3.3, we take the two most common letters in the ciphertext, d and a, and compute the number of times they are adjacent to certain other letters; check that d and a are adjacent to each other 3 times and d and k are adjacent just once. We see that the adjacency of d and a is quite common. We may reasonably guess that one of d and a is a vowel. Because of the presence of the word da, both are not vowels.

Cryptanalysis based on frequencies takes a lot of guesswork and backtracking. Computers are well suited to this task. It is not too difficult to write a program that does the cryptanalysis on such cyphers. One difficulty is that if the ciphertext is short, the frequencies may not correspond to the letter frequencies we expect. Consider a short text like *quick quiz*, which you may send

to a friend regarding the ease of a pop quiz in this class. It will be difficult to decipher this one through frequency analysis, though the pair *qu*, which occurs twice, may help.

Exercise 18

Would it improve security to encrypt the plaintext two times instead of just once? □

3.2.2 Electronic Transmission

For electronic transmissions, the encryption and decryption can be considerably more elaborate than the transposition or substitution codes. The first step in any such transmission is to convert the message to be sent to a bit string, by replacing each symbol of the alphabet by its ascii representation, for instance. This string is now the plaintext. Next, the plaintext is broken up into fixed size blocks, typically around 64 bits in length, which are then encrypted and sent.

A simple encryption scheme is as follows. Alice and Bob agree on a key k, which is a bit string of the same length as the block. Encrypt x by y, where $y = x \oplus k$, i.e., y is the exclusive or of x and k. Bob, on receiving y, computes $y \oplus k$ which is $(x \oplus k) \oplus k$, i.e., x, the original message. Eve can only see y, which appears as pure gibberish. The transmission can only be decrypted by someone in possession of key k.

There are many variations of this simple scheme. It is better to have a long key, much longer than the block length, so that successive blocks are encrypted using different strings. When the bits from k run out, wrap around and start reusing the bits of k from the beginning. Using a longer key reduces the possibility of the code being broken.

This communication scheme is simple to program; in fact, encryption and decryption have the same program. Each operation is fast, requiring time proportional to a block length for encryption (and decryption). Yet, the scheme has significant drawbacks. If Eve can decode a single block, she can decode all blocks (assuming that the key length is the same as the block length), because given x and y where $y = x \oplus k$, k is simply $x \oplus y$. Also, Alice and Bob will have to agree on a key before the transmission takes place, so the keys have to be transmitted first in a secure manner, a problem known as *key exchange*. For these and other reasons, this form of encryption is rarely used in high security applications.

A major problem in devising a secure communication protocol is that Alice may send several messages to Bob, and as the number of transmissions increase, so is the probability of breaking the code. Therefore, it is advisable to use a different key for each transmission. This idea can be implemented as follows. Alice and Bob possess a common sequence of distinct keys, called a *pad*. Each key in the pad is used for exactly one message transmission. Both parties discard a key after it has been used; so, the pad is a *one-time pad*.

It can be shown that one-time pads are unbreakable. However, the major difficulty is in sharing the pad. How can Alice and Bob agree on a pad to begin

with? In ancient times it was conceivable that they could agree on a common book —say, the King James version of the Bible— and use successive strings from the book as keys. However, the need to develop different pads for each pair of communicants, and distribute the pads efficiently (i.e., electronically) and securely, makes this scheme impractical.

Many military victories, defeats and political intrigues over the entire course of human history are directly attributable to security/breakability of codes. Lively descriptions appear in a delightful book by Singh [44].

3.3 Public Key Cryptography

The coding schemes given in the last section were symmetric, in the sense that given the encryption mechanism it is easy to see how to decrypt a message. Thus Alice and Bob, the sender and the receiver, both share the same secret, the key. A consequence of this observation is that the key has to be transmitted before any data can be transmitted.

A novel idea is for Bob to publicly announce a function f that is to be used to encrypt any plaintext to be sent to him, i.e., Alice should encrypt x to $f(x)$ and then send the latter to Bob. Function f is the public key of Bob. Upon receiving the message, Bob applies the inverse function f^{-1}, a private key, thus obtaining $f^{-1}(f(x))$, i.e., x. Eve knows f; so, theoretically, she can also compute x. However, f is chosen so that it is *computationally intractable* to deduce f^{-1} given only f, that is, the computation will take an extraordinarily long time before Eve can deduce f^{-1}. Bob, who designed f, also designed f^{-1} simultaneously. So, he can decrypt the message.

Let us examine some of the implications of using public key cryptography. First, there is no need to exchange any key, because there are no shared secrets. There is a publicly open database in which every one posts his own public key. Any one may join or drop out of this community at any time. Alice sends a message to Bob by first reading his key, f, from the database, applying f to the plaintext x and then sending $f(x)$ to him. Eve can see who sent the message (Alice), who will receive the message (Bob), the public key of the recipient (f) and the ciphertext ($f(x)$). Yet, she is powerless, because it will take her eons to decode this message. Note that Alice also cannot decrypt any message sent to Bob by another party.

Function f is called *one-way* because it is easy to apply —$f(x)$ can be computed easily from x— though hard to invert, that is, to compute x from $f(x)$ without using additional information. Let me repeat that it is theoretically possible to compute x given f and $f(x)$; simply try all possible messages y of appropriate length as candidates for x, compute $f(y)$, and then compare it against $f(x)$. This is not practical for any but the very shortest x because of the number of possible candidates. If the function f is well-chosen, Eve has no other way of decrypting the message. If the message is 64 bits long and she can check 10^{10} messages a second, it will still take around 58 years to check all possible messages. She could, of course, be lucky, and get a break early in the

search; but, the probability of being lucky is quite low.

The notion of computational intractability was not available to the early cryptanalysts; it is a product of modern computer science. We now know that there are certain problems which, though decidable, are computationally intractable in that they take huge amounts of time to solve. Normally, this is an undesirable situation. We have, however, turned this disadvantage to an advantage by putting the burden of solving an intractable problem on Eve, the eavesdropper.

The idea of public key cryptography using one-way functions is due to Diffie and Hellman [13]. Rivest, Shamir and Adelman [41] were the first to propose a specific one-way function that has remained unbroken (or, so it is believed). In the next section, I develop the theory behind this one-way function.

3.3.1 Mathematical Preliminaries

Modular Arithmetic

Henceforth, all variables are positive integers unless stated otherwise. We write "$x \bmod n$" for the remainder of x divided by n. Two integers x and y that have the same remainder after division by n are said to be *congruent* mod n; in that case, we write

$$x \overset{\text{mod } n}{\equiv} y$$

That is,

$$(x \overset{\text{mod } n}{\equiv} y) \equiv (x \bmod n = y \bmod n)$$

So, $x \overset{\text{mod } n}{\equiv} y$ means $x - y$ is divisible by n.

Note that congruence (mod n) is an equivalence relation over integers. Below, we list a few properties of the congruence relation. Variables u, v, p, x and y are positive integers.

- (P1)

$$\frac{\begin{array}{r} u \overset{\text{mod } p}{\equiv} v, \\ x \overset{\text{mod } p}{\equiv} y \end{array}}{\begin{array}{l} u + x \overset{\text{mod } p}{\equiv} v + y, \\ u - x \overset{\text{mod } p}{\equiv} v - y, \\ u \times x \overset{\text{mod } p}{\equiv} v \times y \end{array}}$$

- (P2; Corollary) For all n, $n \geq 0$,

$$\frac{x \overset{\text{mod } p}{\equiv} y}{x^n \overset{\text{mod } p}{\equiv} y^n}$$

- (Modular Simplification Rule) Let e be any expression over integers that has only addition, subtraction, multiplication and exponention as its operators. Let e' be obtained from e by replacing any subexpression t of e by $(t \bmod p)$. Then, $e \overset{\text{mod } p}{\equiv} e'$, i.e., $e \bmod p \;=\; e' \bmod p$.

  Note that an exponent is not a subexpression; so, it can't be replaced by its mod.

Examples

$(20 + 5) \bmod 3 = ((20 \bmod 3) + 5) \bmod 3$
$((x \times y) + g) \bmod p = (((x \bmod p) \times y) + (g \bmod p)) \bmod p$
$x^n \bmod p = (x \bmod p)^n \bmod p$
$x^{2n} \bmod p = (x^2)^n \bmod p = (x^2 \bmod p)^n \bmod p$
$x^n \bmod p = x^{n \bmod p} \bmod p$, is wrong. □

Relatively Prime Positive integers x and y are relatively prime iff $\gcd(x, y) = 1$. By convention, 0 and x are relatively prime iff $x = 1$. Note that $\gcd(0, 0)$ is undefined.

- (P3) For p and q relatively prime,
  $\langle u \overset{\text{mod } p}{\equiv} v \;\wedge\; u \overset{\text{mod } q}{\equiv} v\rangle \;\equiv\; \langle u \overset{\text{mod } p\times q}{\equiv} v\rangle$

Exercise 19
Disprove each of the following conclusions.

$$\frac{\begin{array}{r} u \overset{\text{mod } p}{\equiv} v, \\ x \overset{\text{mod } p}{\equiv} y \end{array}}{\begin{array}{r} \max(u, x) \overset{\text{mod } p}{\equiv} \max(v, y), \\ u^x \overset{\text{mod } p}{\equiv} v^y \end{array}}$$

Solution Use $p = 3$, $u, v = 2, 2$ and $x, y = 4, 1$. □

The following rule allows us to manipulate exponents, which we can't do using only the modular simplification rule (see the previous exercise).

- (P4; due to Fermat) $b^{p-1} \bmod p = 1$, where p is prime, and b and p are relatively prime.

Exercise 20
With b and p as in (P4) show that for any nonnegative integer m

$$b^m \overset{\text{mod } p}{\equiv} b^{m \bmod (p-1)}$$

Write b^m as $b^{p-1} \times b^{p-1} \ldots \times b^{m \bmod (p-1)}$. Use (P4) to reduce each $b^{p-1} \bmod p$ to 1. □

Extended Euclid Algorithm

Given nonnegative integers x and y, where both integers are not zero, there exist integers a and b such that

$$a \times x + b \times y = \gcd(x, y).$$

Note that a and b need not be positive, nor are they unique. This result can be proven by giving a procedure to compute a and b.

The classical Euclid algorithm for computing the gcd is as follows.

$u, v := x, y$
$\{u \geq 0,\ v \geq 0,\ u \neq 0\ \vee\ v \neq 0,\ \gcd(x, y) = \gcd(u, v)\}$
while $v \neq 0$ **do**
  $u, v := v, u \bmod v$
od
$\{\gcd(x, y) = \gcd(u, v),\ v = 0\}$
$\{\gcd(x, y) = u\}$

One way of computing $u \bmod v$ is to explicitly compute the quotient q, $q = \lfloor u/v \rfloor$, and subtract $v \times q$ from u, as shown below.

$u, v := x, y$
$\{u \geq 0,\ v \geq 0,\ u \neq 0\ \vee\ v \neq 0,\ \gcd(x, y) = \gcd(u, v)\}$
while $v \neq 0$ **do**
  $q := \lfloor u/v \rfloor$;
  $u, v := v, u - v \times q$
od
$\{\gcd(x, y) = u\}$

To compute a and b as required, we augment this program by introducing variables a, b and another pair of variables c, d, which satisfy the invariant

$$(a \times x + b \times y = u)\ \wedge\ (c \times x + d \times y = v)$$

An outline of the program is shown below.

$u, v := x, y;\ a, b := 1, 0;\ c, d := 0, 1;$
while $v \neq 0$ **do**
  $q := \lfloor u/v \rfloor$;
  α: $\{(a \times x + b \times y = u)\ \wedge\ (c \times x + d \times y = v)\}$
  $u, v := v, u - v \times q$;
  $a, b, c, d := a', b', c', d'$
  β: $\{(a \times x + b \times y = u)\ \wedge\ (c \times x + d \times y = v)\}$
od

The remaining task is to calculate a', b', c', d' so that the given annotations are correct, i.e., the invariant $(a \times x + b \times y = u)\ \wedge\ (c \times x + d \times y = v)$ holds at program point β. Using backward substitution, we need to show that the following proposition holds at program point α.

$$(a' \times x + b' \times y = v) \ \wedge \ (c' \times x + d' \times y = u - v \times q)$$

We are given that the proposition $(a \times x + b \times y = u) \ \wedge \ (c \times x + d \times y = v)$ holds at α. Therefore, we may set

$$a', b' = c, d$$

Now, we compute c' and d'.

$$\begin{array}{ll} & c' \times x + d' \times y \\ = & \{\text{from the invariant}\} \\ & u - v \times q \\ = & \{a \times x + b \times y = u \text{ and } c \times x + d \times y = v\} \\ & (a \times x + b \times y) - (c \times x + d \times y) \times q \\ = & \{\text{algebra}\} \\ & (a - c \times q) \times x + (b - d \times q) \times y \end{array}$$

So, we may set

$$c', d' = a - c \times q, b - d \times q$$

The complete algorithm is:

$u, v := x, y;\ a, b := 1, 0;\ c, d := 0, 1;$
while $v \neq 0$ **do**
  $q := \lfloor u/v \rfloor;$
  $\alpha: \ \{(a \times x + b \times y = u) \ \wedge \ (c \times x + d \times y = v)\}$
  $u, v := v, u - v \times q;$
  $a, b, c, d := c, d, a - c \times q, b - d \times q$
  $\beta: \ \{(a \times x + b \times y = u) \ \wedge \ (c \times x + d \times y = v)\}$
od

At the termination of the algorithm,

$$\begin{array}{ll} & a \times x + b \times y \\ = & \{\text{from the invariant}\} \\ & u \\ = & \{u = \gcd(x, y), \text{ from the annotation of the first program in page 54}\} \\ & \gcd(x, y) \end{array}$$

Example Let $x, y = 157, 2668$. In Table 3.4, we show the steps of the extended Euclid algorithm. □

Exercise 21

Show that the given algorithm terminates. Also, prove that $\alpha : \{(a \times x + b \times y = u) \ \wedge \ (c \times x + d \times y = v)\}$ is a loop invariant. Use the annotations shown in the program. □

a	b	u	c	d	v	q
1	0	157	0	1	2668	
						0
0	1	2668	1	0	157	
						16
1	0	157	-16	1	156	
						1
-16	1	156	17	-1	1	
						156
17	-1	1	-2668	157	0	

Table 3.4: Computation with the extended Euclid algorithm

Exercise 22
Prove using induction that for any pair of positive integers x and y, there exist integers a and b such that

$$a \times x + b \times y = \gcd(x, y).$$

Order pairs of integers lexicographically. Develop an alternative proof by using induction over pairs of integers ordered according to the magnitude of their sums. □

3.3.2 The RSA Scheme

Joining the Cryptosystem

A principal, such as Bob, joins the public key cryptosystem by following the steps given below.

- Choose two large primes p and q. There are efficient probabilistic schemes for computing p and q.
- Let $n = p \times q$. And, define $\phi(n) = (p-1) \times (q-1)$. Any message below n in value can be encrypted.
- Choose an integer d, $1 \leq d < n$, where d and $\phi(n)$ are relatively prime. In particular, any prime exceeding $\max(p, q)$ is a valid choice for d.
- Find e such that $d \times e \overset{\text{mod } \phi(n)}{\equiv} 1$.
  Computation of e is based on the Extended Euclid algorithm of page 54. Set $x := d$ and $y := \phi(n)$ in the formula $a \times x + b \times y = \gcd(x, y)$ to get:

$$\begin{array}{ll} & a \times d + b \times \phi(n) = \gcd(d, \phi(n)) \\ \Rightarrow & \{d \text{ and } \phi(n) \text{ are relatively prime, from the choice of } d\} \\ & a \times d + b \times \phi(n) = 1 \\ \Rightarrow & \{\text{definition of mod}\} \\ & a \times d \overset{\text{mod } \phi(n)}{\equiv} 1 \end{array}$$

Now, let e be a. If e is positive, we are done. If e is negative (it can't be zero) add $\phi(n)$ to it enough times to make it positive. Note that $(a + k \times \phi(n)) \times d \overset{\text{mod } \phi(n)}{\equiv} 1$, for any k.

- At this stage, Bob has the following variables:
  1. p and q, which are primes,
  2. n, which is $p \times q$, and $\phi(n)$, which is $(p-1) \times (q-1)$,
  3. d, which is a positive integer smaller than n and relatively prime to $\phi(n)$, and
  4. e, where $d \times e \overset{\text{mod } \phi(n)}{\equiv} 1$.
- Publicize (e, n) as the public key. Save (d, n) as the private key.

Example Let $p = 47$ and $q = 59$. Then, $n = p \times q = 2773$, and $\phi(2773) = (47-1) \times (59-1) = 2668$. Let $d = 157$. Now, e is computed as shown in Table 3.4 of page 56. Thus, $e = 17$. Verify that $17 \times 157 \overset{\text{mod } \phi(n)}{\equiv} 1$. □

Encryption

To send message M to a principal whose public key is (e, n) and $0 \leq M < n$, send M' where

$$M' = (M^e \bmod n)$$

Example; contd. Let us represent each letter of the alphabet by two digits, with

white space = 00 a = 01 b = 02, etc.

Suppose the message to be sent is "bad day". The representation yields: 02010400040125.

Since n is 2773, we can convert any pair of letters to a value below n, the largest such pair being zz which is encoded as 2626. Therefore, our block length is 2 letters. We get the following blocks from the encoded message: 0201 0400 0401 2500; we have appended an extra blank at the end of the last block to make all blocks have equal size.

Now for encryption of each block. We use the parameters from the previous example, where $e = 17$. For the first block, we have to compute $0201^{17} \bmod 2773$, for the second $0400^{17} \bmod 2773$, etc. □

There is an efficient way to raise a number to a given exponent. To compute M^{17}, we need not multiply M with itself 16 times. Instead, we see that $M^{17} = M^{16} \times M = (M^8)^2 \times M = ((M^4)^2)^2 \times M = (((M^2)^2)^2)^2 \times M$. The multiplication strategy depends on the binary representation of the exponent. Also, at each stage, we may apply $\bmod n$, so that the result is always less than n. Specifically, we use

$M^{2t} \bmod n = (M^t \bmod n)^2 \bmod n$
$M^{2t+1} \bmod n = ((M^t \bmod n)^2 \times M) \bmod n$

The following algorithm implements this strategy. Let e, the exponent, in binary be: $e_k\ e_{k-1} \ldots e_0$. For $e = 17$, we get 10001. Next, use the following algorithm that looks at the bits of e from the higher to the lower order; the result of encryption is in C. The loop invariant is: $C = M^h \bmod n$, where h is the portion of the exponent seen so far, i.e., $e_k\ e_{k-1} \ldots e_i$ (initially, $h = 0$).

$C := 1;$
for $i = k..0$ **do**
  if $e_i = 0$
    then $C := C^2 \bmod n$
    else $C := ((C^2 \bmod n) * M) \bmod n$
  fi
od

We encrypt 0201 to 2710 and 0400 to 0017.

Exercise 23

The algorithm to compute $M^e \bmod n$, given earlier, scans the binary representation of e from left to right. It is often easier to scan the representation from right to left, because we can check if e is even or odd easily on a computer. We use:

$M^{2t} = (M^2)^t$
$M^{2t+1} = (M^2)^t \times M$

Here is an algorithm to compute M^e in C. Prove its correctness using the loop invariant $C * m^h = M^e$. Also, modify the algorithm to compute $M^e \bmod n$.

$C := 1;\ h, m := e, M;$
while $h \neq 0$ **do**
  if $odd(h)$ **then** $C := C * m$ **fi** ;
  $h := h \div 2;$
  $m := m^2$
od
$\{C = M^e\}$

Decryption

On receiving an encrypted message M', $0 \leq M' < n$, Bob, whose private key is (d, n), computes M'' as follows.

$M'' = (M'^d \bmod n)$

We show below that $M'' = M$.

Example We continue with the previous example. The encryption and decryption steps are identical, except for different exponents. We use the encryption algorithm with exponent 157 to decrypt. The encryption of 0201 is 2710 and of 0400 is 0017. Computing $2710^{157} \bmod 2773$ and $0017^{157} \bmod 2773$ yield the original blocks, 0201 and 0400. □

Lemma 1: For any M, $0 \leq M < n$, $M^{d \times e} \overset{\bmod p}{\equiv} M$.

Proof:

$$
\begin{array}{ll}
 & M^{d \times e} \bmod p \\
= & \{d \times e \overset{\bmod \phi(n)}{\equiv} 1, \text{ and } \phi(n) = (p-1) \times (q-1)\} \\
 & M^{t \times (p-1)+1} \bmod p, \text{ for some } t \\
= & \{\text{rewriting}\} \\
 & ((M^{(p-1)})^t \times M) \bmod p \\
= & \{\text{modular simplification: replace } (M^{(p-1)}) \text{ by } (M^{(p-1)}) \bmod p\} \\
 & ((M^{(p-1)} \bmod p)^t \times M) \bmod p \\
= & \{\text{Consider two cases:} \\
 & \quad \bullet\ M \text{ and } p \text{ are not relatively prime:} \\
 & \quad \text{Since } p \text{ is prime, } M \text{ is a multiple of } p, \text{ i.e.,} \\
 & \quad (M \bmod p) = 0. \text{ So, } M^{(p-1)} \bmod p = 0. \\
 & \quad \text{The entire expression is 0, thus equal to } M \bmod p. \\
 & \quad \bullet\ M \text{ and } p \text{ are relatively prime:} \\
 & \quad \text{Then, } (M^{(p-1)}) \bmod p = 1, \text{ from (P4).} \\
 & \quad \text{The expression is } (1^t \times M) \bmod p = M \bmod p. \\
 & \} \\
 & M \bmod p
\end{array}
$$

Lemma 2: For any M, $0 \leq M < n$, $(M^{d \times e} \bmod n) = M$.

Proof:

$$
\begin{array}{lll}
M & \overset{\bmod p}{\equiv} & M^{d \times e} & \text{, from Lemma 1} \\
M & \overset{\bmod q}{\equiv} & M^{d \times e} & \text{, replacing } p \text{ by } q \text{ in Lemma 1} \\
M & \overset{\bmod n}{\equiv} & M^{d \times e} & \text{, from above two, using P3 and } n = p \times q \\
(M \bmod n) & = & (M^{d \times e} \bmod n) & \text{, from above} \\
M & = & (M^{d \times e} \bmod n) & \text{, } M < n \text{; so } M \bmod n = M
\end{array}
$$

Theorem: $M'' = M$.

Proof:

$$
\begin{array}{ll}
 & M'' \\
= & \{\text{definition of } M'', \text{ from the decryption step}\} \\
 & M'^d \bmod n \\
= & \{\text{definition of } M', \text{ from the encryption step}\}
\end{array}
$$

$$
\begin{array}{ll}
 & (M^e \bmod n)^d \bmod n \\
= & \{\text{apply modular simplification to } x^d\text{; replace } x \text{ by } x \bmod n. \\
 & \quad (x^d \bmod n) \;=\; (x \bmod n)^d \bmod n \\
 & \quad \text{replace } x \text{ by } M^e \\
 & \quad (M^e)^d \bmod n \;=\; (M^e \bmod n)^d \bmod n \\
 & \} \\
 & (M^e)^d \bmod n \\
= & \{(M^e)^d \bmod n = M^{d\times e} \bmod n\text{; apply Lemma 2}\} \\
 & M
\end{array}
$$

Breaking RSA is hard, probably

The question of breaking RSA amounts to extracting the plaintext from the ciphertext. Though there is no proof for it, it is strongly believed that in order to break RSA, you have to compute the the private key given the public key. That is, given (e, n), find d where $d \times e \overset{\bmod \phi(n)}{\equiv} 1$. We show that computing d is as hard as factoring n.

It is strongly believed that factoring a large number is intractable. In the naive approach to factoring, we have to test all numbers at least up to $\sqrt{n}$ to find a factor of n. If n is a 200 digit number, say, approximately 10^{100} computation steps are needed. The best known algorithm for factoring a 200 digit number would take about a million years. We can speed up matters by employing supercomputers and a whole bunch of them to work in parallel. Yet, it is unlikely that factoring would be done fast enough to justify the investment. So, it is strongly believed —though not proven— that RSA is unbreakable.

Next, we show that computing d is easy given the factors of n. Suppose we have factored n into primes p and q. Then, we can compute $\phi(n)$, which is $(p-1) \times (q-1)$. Next, we can compute d as outlined earlier (we have shown how to compute e from d; computing d from e follows the same steps with the roles of d and e reversed).

The proof in the other direction, —if we have d and e where $d \times e \overset{\bmod \phi(n)}{\equiv} 1$, then we can factor n— is more technical. It can be shown that n is easily factored given any multiple of $\phi(n)$, and $(d \times e) - 1$ is a multiple of $\phi(n)$.

An easier result is that n can be factored if $\phi(n)$ is known. Recall that $\phi(n) = (p-1) \times (q-1)$ and $n = p \times q$. Hence, $\phi(n) = n - (p+q) + 1$. From n and $\phi(n)$, we get $p \times q$ and $p+q$. Observe that $p - q = \sqrt{(p-q)^2} = \sqrt{(p+q)^2 - 4 \times p \times q}$. Therefore, $p - q$ can be computed. Then, $p = \frac{(p+q)+(p-q)}{2}$ and $q = \frac{(p+q)-(p-q)}{2}$.

3.4 Digital Signatures

Public key cryptography neatly solves a related problem, affixing a digital signature to a document. Suppose Bob receives a message from someone claiming to be Alice; how can he be sure that Alice sent the message? To satisfy Bob, Alice affixes her signature to the message, as described below.

Alice encrypts the message using her *private key*; this is now a signed message. If the message is intended for Bob's eyes only, she encrypts the signed message with Bob's public key. Bob first decrypts the message using his own private key, then decrypts the signed message using Alice's public key. More formally, let x be the message, f_a and f_b the public keys of Alice and Bob, and f_a^{-1} and f_b^{-1} be their private keys, respectively. Then $f_a^{-1}(x)$ is the message signed by Alice. She may send $f_b(f_a^{-1}(x))$, encrypting it for Bob's eyes. Alice may include her name in plaintext, $f_b(\text{“alice”} + f_a^{-1}(x))$ where $+$ is concatenation, so that Bob will know whose public key he should apply to decrypt the signed message.

We show that such signatures satisfy a number of desirable properties. First, decrypting any message with the Alice's public key will result in gibberish unless it has been encrypted with her private key. So, if Bob is able to get a meaningful message by decryption, he is convinced that Alice sent the message.

Second, Alice cannot deny sending the message, because no one else has access to her private key. An impartial judge can determine that Alice's signature appears on the document (message) by decrypting it with her public key. Note that the judge does not need access to any private information.

Third, no one can modify this message while keeping Alice's signature affixed to it. Thus, no electronic cutting and pasting of the message/signature is possible.

Observe a very important property of the RSA scheme: any message can be encrypted by the public or the private key and decrypted by its inverse.

Digital signatures are now accepted for electronic documents. A user can sign a check, or a contract, or even a document that has been signed by other parties.

Another look at one-time pads We know that one-time pads provide complete security. The only difficulty with them is that both parties to a transmission must have access to the same pad. We can overcome this difficulty using RSA, as follows.

Regard each one-time pad as a random number. Both parties to a transmission have access to a pseudo-random number generator which produces a stream of random numbers. The pseudo-random number generator is public knowledge, but the seed which the two parties use is a shared secret. Since they use the same seed, they will create the same stream of random numbers. Then the encryption can be relatively simple, like taking exclusive or.

This scheme has one drawback, having the seed as a shared secret. RSA does not have this limitation. We can use RSA to establish such a secret: Bob generates a random number and sends it to Alice, encrypted by Alice's public key. Then, both Bob and Alice know the seed.

Security of communication with a trusted third party We have so far assumed that all public keys are stored in a public database and Alice can query the database manager, David, to get Bob's public key (in plaintext). Suppose

Eve intercepts the message sent by David to Alice, and replaces the public key of Bob by her own. Then Alice encrypts a message for Bob by Eve's public key. Any message she sends to Bob could be intercepted and decrypted by Eve.

The problem arises because Alice does not know that the message received from David is not authentic. To establish authenticity, David —often called a *trusted third party*— could sign the message. Then, Eve cannot do the substitution as described above.

Trusted third parties play a major role in security protocols, and authenticating such a party is almost always handled by digital signatures.

Acknowledgement I am thankful to Steve Li for simplifying one of the proofs.

Chapter 4

Finite State Machines

4.1 Introduction

4.1.1 Wolf-Goat-Cabbage Puzzle

A shepherd arrives at a river bank with a wolf, a goat and a cabbage. There is a boat there that can carry them to the other bank. However, the boat can carry the shepherd and at most one other item. The shepherd's actions are limited by the following constraints: if the wolf and goat are left alone, the wolf will devour the goat, and if the goat and the cabbage are left alone, well, you can imagine...

You can get a solution quickly by rejecting certain obvious possibilities. But let us attack this problem more systematically. What is the state of affairs at any point during the passage: what is on the left bank, what is on the right bank, and where the boat is (we can deduce the contents of the boat by determining which items are absent from both banks). The state of the left bank is a subset of {w,g,c} —w for wolf, g for goat, and c for cabbage— and similarly for the right bank. The shepherd is assumed to be with the boat (the cabbage cannot steer the boat :-)), so the state of the boat is that it is: (1) positioned at the left bank, (2) positioned at the right bank, (3) in transit from left to right, or (4) in transit from right to left; let us represent these possibilities by the symbols, L, R, LR, RL, respectively.

Thus, we represent the initial state by a triple like ⟨{w,g,c}, L, {}⟩. Now what possible choices are there? The shepherd can row alone, or take one item with him in the boat, the wolf, the goat or the cabbage. These lead to the following states respectively.

⟨{w,g,c}, LR, {}⟩
⟨{g,c}, LR, {}⟩
⟨{w,c}, LR, {}⟩
⟨{w,g}, LR, {}⟩

Observe that all states except ⟨{w,c}, LR, {}⟩ are inadmissible, since someone will consume something. So, let us continue the exploration from ⟨{w,c}, LR, {}⟩.

When the shepherd reaches the other bank, the state changes from ⟨{w,c}, LR, {}⟩ to ⟨{w,c}, R, {g}⟩. Next, the shepherd has a choice: he can row back with the goat to the left bank (an obviously stupid move, because he will then be at the initial state), or he may row alone. In the first case, we get the state ⟨{w,c}, RL, {}⟩, and in the second case ⟨{w,c}, RL, {g}⟩. We may continue exploring from each of these possibilities, adding more states to the diagram. Figure 4.1 shows the initial parts of the exploration more succinctly.

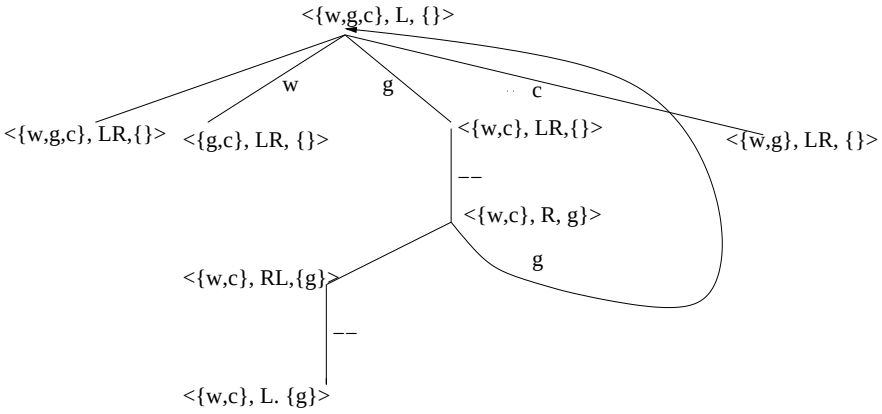


Figure 4.1: Partial State Space for the Wolf-Goat-Cabbage Problem

The important thing to note is that the number of states is finite (prove it). So, the exploration will terminate sometime.

Exercise 24
Complete the diagram. Show that a path in the graph corresponds to a solution. How many solutions are there? Can you define states differently to derive a smaller diagram? □

Remark A beautiful treatment of this puzzle appears in Dijkstra [15]. He shows that with some systematic thinking you can practically eliminate the state-space search. You can play the game at `http://www.plastelina.net`; choose game 1. □

4.1.2 A Traffic Light

A traffic light is in one of three states, green, yellow or red. The light changes from green to yellow to red; it cannot change from green to red, red to yellow or yellow to green. We may depict the permissible state transitions by the diagram shown in Figure 4.2.

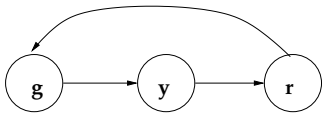


Figure 4.2: State Transitions in a Traffic Light

What causes the state transitions? It is usually the passage of time; let us say that the light changes every 30 seconds. We can imagine that an internal clock generates a pulse every 30 seconds that causes the light to change state. Let symbol p denote this pulse.

Suppose that an ambulance arrives along an intersecting road and remotely sets this light red (so that it may proceeed without interference from vehicles travelling along this road). Then, we have a new state transition, from green to red and from yellow to red, triggered by the signal from the ambulance; call this signal a. See Figure 4.3 for the full description.

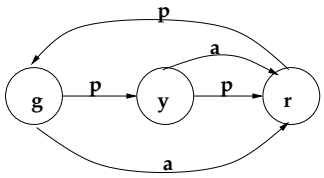


Figure 4.3: State Transitions in an Enhanced Traffic Light

4.1.3 A Pattern Matching Problem

You are given a list of English words, as in a dictionary. Find the words in which the five vowels —a,e,i,o,u— are in order. These are words like "abstemious", "facetious" and "sacrilegious". But not "tenacious", which contains all the vowels but not in order.

Let us design a program to solve this problem. Our program looks at each word in the dictionary in turn. For each word it scans it until it finds an "a", or fails to find it. In the latter case, it rejects the word and moves on to the next word. In the first case, it resumes its search from the point where it found "a" looking for "e". This process continues until all the vowels in order are found, or the word is rejected.

A programming hint: Sentinel How do you search for a symbol c in a string $S[0..N]$? Here is the typical strategy.

$i := 0$;
while $S[i] \neq$ "c" $\wedge\ i \leq N$ **do**

$i := i + 1$
od ;
if $i \leq N$ **then** "success" **else** "failure" **fi**

A simpler strategy uses a "sentinel", an item at the end of the list which guarantees that the search will not fail. It simplifies AND speeds up the loop.

$S[N+1] :=$ "c"; $i := 0$;
while $S[i] \neq$ "c" **do**
$i := i + 1$
od ;
if $i \leq N$ **then** "success" **else** "failure" **fi** □

Bonus Programming Exercise Write a program for the pattern matching problem, and apply it to a dictionary of your choice. □

If you complete the program you will find that its structure is a mess. There are five loops, each looking for one vowel. They will be nested within a loop. A failure causes immediate exit from the corresponding loop. (Another possibility is to employ a procedure which is passed the word, the vowel and the position in the word where the search is to start.) Modification of this program is messy. Suppose we are interested in words in which exactly these vowels occur in order, so "sacrilegious" will be rejected. How will the program be modified? Suppose we don't care about the order, but we want all the vowels to be in the word; so, "tenacious" will make the cut. For each of these modifications, the program structure will change significantly.

What we are doing in all these cases is to match a pattern against a word. The pattern could be quite complex. Think about the meaning of pattern if you are searching a database of music, or a video for a particular scene. Here are some more examples of "mundane" patterns that arise in text processing.

Examples

1. *integer*: An *integer* is either 0 or a non-zero digit followed by any number of digits. So, 0, 31, 310 are all integers , but 00, 03, 3.7 are not.

2. *signed integer*: An integer with an optional sign, + or −, at its front.

3. *number*: A *number* is either an *integer* or an *integer* followed by a period followed by any number of digits.

4. *floating point number*: A *number* with an optional sign, + or −, at its front, and an optional mantissa that is of the form $E\ n$, where n is a signed integer. The following are floating point numbers.

$$+3.0,\ 3.0,\ 0.3E1,\ 0.3E+1,\ -0.3E+1,\ -3E8$$

begin	**end**	**while**	**do**	**od**	**if**	**then**	**fi**
106	107	100	101	102	103	104	105

Table 4.1: Translations of keywords

:=	;	<	≤	=	≠	>	≥	+
1000	1001	1002	1003	1004	1005	1006	1007	1008

Table 4.2: Translations of non-keywords

These definitions are not the standard ones. For instance, while we accept 31.2 and 0.0 and 30.02 as numbers, .3 is not a number.

Here are some more examples of patterns.

1. Any string ending in a white space. This is often called a *word*.
2. Any string in which "(" and ")" are balanced.
3. Any string starting with "b" followed by any number of "a"s and then a "d". These strings are: "bd", "bad", "baad", "baaad", ... □

In all cases, we are dealing with strings —a sequence of symbols— drawn from a fixed alphabet. A string may or may not satisfy a pattern: "abstemious" satisfies the pattern of having all five vowels in order, and .3 does not satisfy the pattern of being a number.

A longer pattern matching example Consider a language that has the following keywords:

begin end while do od if then fi

A lexical processor for the program may have to:

1. Convert every keyword to a number, as described in Table 4.1.
2. Convert every non-keyword to a distinct 2-digit number,
3. Convert every other symbol as described in Table 4.2, and
4. Ignore comments (the stuff that appears between braces) and extra white spaces.

Thus, a string like

while $i \neq n$ **do** {silly loop} $j := i + 1$ **od**

will be converted as shown in Table 4.3.

while	i	$\neq$	n	**do**	{silly loop}	j	:=	i	+	1	**od**
100	10	1005	11	101		12	1000	10	1008	13	102

Table 4.3: Translation of a program

4.2 Finite State Machine

4.2.1 What is it?

Consider the original problem of checking a word for vowels in order. We can describe a machine to do the checking as shown in Figure 4.4. The machine has six states, each shown as a circle. Each directed edge has a label, the name of a symbol (or set of symbols) from a specified *alphabet*.

The machine operates as follows. Initially, the machine is in the state to which the "start" arrow points (the state labeled 1). It receives a stream of symbols. Depending on the symbol and its current state the machine determines its next state, which may be the same as the current state. Thus, in state 1, if it receives symbol "a" it transits to state 2, and otherwise (shown by the arrow looping back to the state) it stays in state 1. Any state shown by a double circle is called an *accepting state*; the remaining states are *rejecting states*. In this example, the only accepting state is 6.

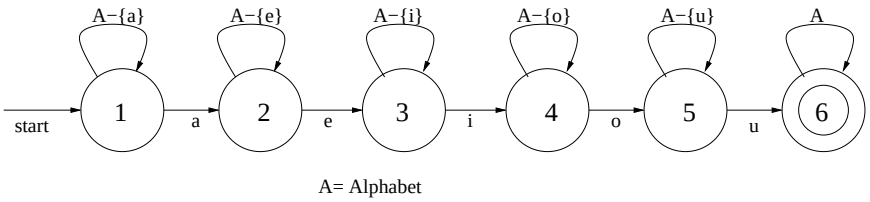


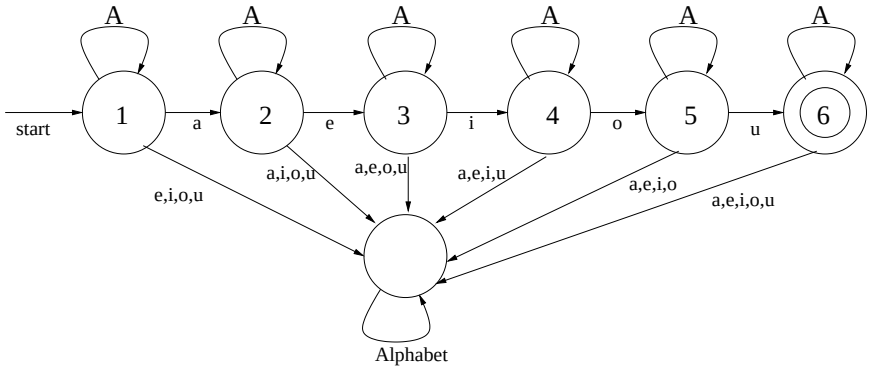
Figure 4.4: Machine to check for vowels in order

If the machine in Figure 4.4 receives the string "abstemious" then its successive states are: 1 2 2 2 2 3 3 4 5 6 6. Since its final state is an accepting state, we say that the string is *accepted* by the machine. A string that makes the machine end up in a rejecting state is said to be *rejected* by the machine.

Which state does the machine end up in for the following strings: aeio, tenacious, f, aaeeiioouu, ϵ (ϵ denotes the empty string)? Convince yourself that the machine accepts a string iff five vowels appear in order in that string.

Exercise 25

Draw a machine that accepts strings which contain the five vowels in order, and no other vowels. So, the machine will accept "abstemious", but not "sacrilegious". See Figure 4.5. □



A = Alphabet − {a,e,i,o,u}

Figure 4.5: Machine to check for exactly five vowels in order

Definitions A (deterministic) finite state machine over a given alphabet has a *finite* number of states, one state designated as the *initial* state, a subset of states designated as *accepting* and a *state transition function* that specifies the next state for each state and input symbol. The machine *accepts* or *rejects* every *finite* string over its alphabet.

Note There is a more general kind of finite state machine called a *nondeterministic* machine. The state transitions are not completely determined by the current state and the input symbol as in the deterministic machines you have seen so far. The machine is given the power of clairvoyance so that it chooses the next state, out of a possible set of successor states, which is the "best" state for processing the remaining unseen portion of the string. □

Exercise 26

1. Design finite state machines for the following problems. Assume that the alphabet is $\{0, 1\}$.
   (a) Accept all strings.
   (b) Reject all strings.
   (c) Accept if the string has an even number of 0s.
   (d) Accept if the string has an odd number of 1s.
   (e) Accept if the conditions in both (1c) and in (1d) apply. Can you find a general algorithm to construct a finite state machine from two given finite state machines, where the constructed machine accepts only if both component machines accept? What assumptions do you have to make about the component machines?

(f) Accept if either of the conditions in (1c) and in (1d) apply. Can you find a general algorithm to construct a finite state machine from two given finite state machines, where the constructed machine accepts only if either component machine accepts? What assumptions do you have to make about the component machines?

(g) Accept iff the machine in (1e) does not accept (that is, a string with an odd number of 0s or even number of 1s is accepted). Again, is there a general procedure?

(h) Convince yourself that you cannot design a finite state machine to accept a string that has an equal number of zeros and ones.

2. For the Wolf-Goat-Cabbage puzzle, design a suitable notation to represent each move of the shepherd using a symbol. Then, any strategy is a string. Design a finite state machine that accepts such a string and enters an accepting state if the whole party crosses over to the right bank, intact, and rejects the string otherwise.

3. For the following problems, the alphabet consists of letters (from the Roman alphabet) and digits (Arabic numerals).

(a) Accept if it contains a keyword, as given in Table 4.1.

(b) Accept if it is an *integer*, as defined in page 66. Modify the machine to accept *signed integer*, as defined in page 66.

(c) Accept if it is a *number* as defined in page 66.

(d) Accept a string of digits if they are strictly increasing. So, 345, 069, 2 will be accepted, whereas 32 will be rejected.

(e) Accept if the string is a legal *identifier*: a letter followed by zero or more symbols (a letter or digit).

4. A computer has n 32-bit words of storage. What is the number of states? For a modern computer, n is around 2^{25}. Suppose each state transition takes a nanosecond (10^{-9} second). How long will it take the machine to go through all of its states?

5. Write a program (in C++ or Java) —without reference to finite state machines— that outputs "accept" if the input is a string with an even number of 0s and an odd number of 1s. Next, hand-translate the finite state machine you have designed for this problem in Exercise (1e) into a program. Compare the two programs in terms of length, simplicity, design time, and execution efficiency. □

Exercise 27
Let F be a finite state machine.

1. Design a program that accepts a description of F and constructs a Java program J equivalent to F. That is, J accepts a string as input and prints "accept" or "reject". Assume that your alphabet is {0,1} , and a special symbol, say #, terminates the input string.

2. Design a program that accepts a description of F and a string s and prints "accept" or "reject" depending on whether F accepts or rejects s. □

4.2.2 Reasoning about Finite State Machines

Consider the finite state machine shown in Figure 4.6. We would like to show that the strings accepted by the machine have an even number of 0s and an odd number of 1s. The problem is complicated by the fact that there are loops.

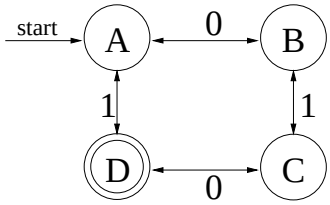


Figure 4.6: Machine that accepts strings with an even number of 0s and an odd number of 1s

The strategy is to guess what string is accepted in each state, and attach that as a label to that state. This is similar to program proving. Let

$p \equiv$ this string has an even number of 0s,
$q \equiv$ this string has an even number of 1s

A plausible annotation of the machine is shown in Figure 4.7. That is, we guess that any string for which the machine state becomes B has an odd number of 0s and an even number of 1s; similarly for the remaining state annotations.

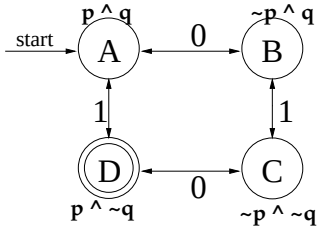


Figure 4.7: Annotation of the machine in Figure 4.6

Verification Procedure The verification procedure consists of three steps: (1) annotate each state with a predicate over finite strings (the predicate defines a set of strings, namely, the ones for which it is *true*), (2) show that the annotation on the initial state holds for the empty string, and (3) for each transition do the following verification: suppose the transition is labeled s and it is from

From A to B: if	$p \wedge q$	holds for x then	$\neg p \wedge q$	holds for $x0$
From A to D: if	$p \wedge q$	holds for x then	$p \wedge \neg q$	holds for $x1$
From B to A: if	$\neg p \wedge q$	holds for x then	$p \wedge q$	holds for $x0$
From B to C: if	$\neg p \wedge q$	holds for x then	$\neg p \wedge \neg q$	holds for $x1$
From C to B: if	$\neg p \wedge \neg q$	holds for x then	$\neg p \wedge q$	holds for $x1$
From C to D: if	$\neg p \wedge \neg q$	holds for x then	$p \wedge \neg q$	holds for $x0$
From D to C: if	$p \wedge \neg q$	holds for x then	$\neg p \wedge \neg q$	holds for $x0$
From D to A: if	$p \wedge \neg q$	holds for x then	$p \wedge q$	holds for $x1$

Table 4.4: Verifications of state transitions

a state annotated with b to one with c; then, show that if b holds for any string x, then c holds for xs.

For the machine in Figure 4.7, we have already done step (1). For step (2), we have to show that the empty string satisfies $p \wedge q$, that is, the empty string has an even number of 0s and 1s, which clearly holds. For step (3), we have to verify all eight transitions, as shown in Table 4.4. For example, it is straightforward to verify the transition from A to B by considering an arbitrary string x with an even number of 0s and 1s ($p \wedge q$) and proving that $x0$ has odd number of 0s and even number of 1s ($\neg p \wedge q$).

Why Does the Verification Procedure Work? It seems that we are using some sort of circular argument, but that is not so. In order to convince yourself that the argument is not circular, construct a proof using induction. The theorem we need to prove is as follows: after processing any string x, the machine state is A, B, C or D iff x satisfies $p \wedge q$, $\neg p \wedge q$, $\neg p \wedge \neg q$ or $p \wedge \neg q$, respectively. The proof of this statement is by induction on the length of x.

For $|x| = 0$: x is an empty string and $p \wedge q$ holds for it. The machine state is A, so the theorem holds.

For $|x| = n + 1$, $n \geq 0$: use the induction hypothesis and the proofs from Table 4.4.

4.2.3 Finite State Transducers

The finite state machines we have seen so far simply accept or reject a string. So, they are useful for doing complicated tests, such as to determine if a string matches a given pattern. Such machines are called *acceptors*. Now, we will enhance the machine so that it also produces a string as output; such machines are called *transducers*. Transducers provide powerful string processing mechanism. Typically, acceptance or rejection of the input string is of no particular importance in transducers; only the construction of the appropriate output string matters.

Pictorially, we will depict a transition as shown in Figure 4.8. It denotes that on reading symbol s, the machine transits from A to B and outputs string

t. The output alphabet of the machine —over which t is a string— may differ from its input alphabet.

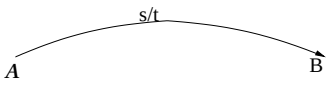


Figure 4.8: Transition Labeling in a Finite State Transducer

Example Accept any string of 0s and 1s. Squeeze each substring of 0s to a single 0 and similarly for the 1s. Thus,

000100110 becomes 01010

A solution is shown in Figure 4.9.

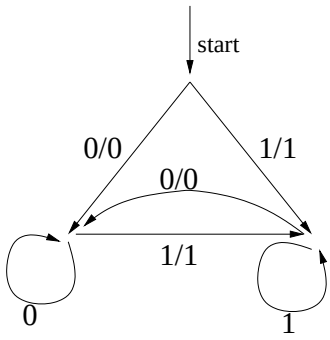


Figure 4.9: Transducer that squeezes each block to a single bit

Verifications of Transducers How do we verify a transducer? We would like to show that the output is a function, f, of the input. For the transducer in Figure 4.9, function f is given by:

$$\begin{array}{lll} f(\epsilon) = \epsilon & f(0) = 0 & f(1) = 1 \\ f(x00) = f(x0) & f(x01) = f(x0)1 & \\ f(x10) = f(x1)0 & f(x11) = f(x1) & \end{array}$$

The verification strategy for finite state acceptors is augmented as follows. As before, annotate each state by a predicate (denoting the set of strings for which the machine enters that state). Show for the initial state that the annotation is satisfied by the empty string and it outputs $f(\epsilon)$. For a transition of the form shown in Figure 4.8, if A is annotated with p and B with q, show that (1) if p holds for any string x, then q holds for xs, and (2) $f(xs) = f(x) +\!\!+ t$

(the symbol $+\!\!+$ denotes concatenation), i.e., the output in state B (which is the output in state A —assumed to be $f(x)$— concatenated with string t) is the desired output for any string for which this state is entered.

Exercise 28

In any string of 0s and 1s replace each 0 by 01 and 1 by 10. □

Exercise 29

The input is a 0-1 string. A 0 that is both preceded and succeeded by at least three 1s is to be regarded as a 1. The first three symbols are to be reproduced exactly. The example below shows an input string and its transformation; the bit that is changed has an overline on it in the input and underline in the output.

$0110111\overline{0}11111000111$ becomes
$0110111\underline{1}11111000111$

Design a transducer for this problem and establish its correctness. □

Solution In Figure 4.10, the transitions pointing downward go to the initial state. Prove correctness by associating with each state a predicate which asserts that the string ending in that state has a certain suffix.

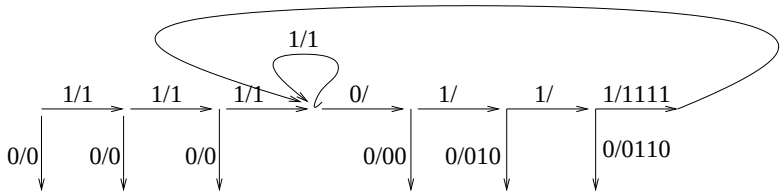


Figure 4.10: Replace 0 by 1 if it is preceded and succeeded by at least three 1s

4.2.4 Serial Binary Adder

Let us build an adding circuit (adder) that receives two binary operands and outputs their sum in binary. We will use the following numbers for illustration.

$$\begin{array}{r} 0\ 1\ 1\ 0\ 0 \\ +\ 0\ 1\ 1\ 1\ 0 \\ \hline 1\ 1\ 0\ 1\ 0 \end{array}$$

The input to the adder is a sequence of bit pairs, one bit from each operand, starting with their lowest bits. Thus the successive inputs for the given example are: (0 0) (0 1) (1 1) (1 1) (0 0). The adder outputs the sum as a sequence of bits, starting from the lowest bit; for this example, the output is 0 1 0 1 1. If

there is a carry out of the highest bit it is not output, because the adder cannot be sure that it has seen all inputs. (How can we get the full sum out of this adder?)

We can design a transducer for this problem as shown in Figure 4.11. There are two states, the initial state is n and the carry state c; in state c, the current sum has a carry to the next position. The transitions are easy to justify. For instance, if the input bits are (0 1) in the n state, their sum is $0 + 1 + 0 = 1$; the last 0 in the sum represents the absence of carry in this state. Therefore, 1 is output and the machine remains in the n state. If the machine is in c state and it receives (0 0) as input, the sum is $0 + 0 + 1 = 1$; hence, it outputs 1 and transits to the n state. For input (1 1) in the c state, the sum is $1 + 1 + 1 = 3$, which is 11 in binary; hence 1 is output and the machine remains in the c state.

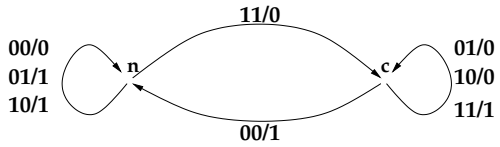


Figure 4.11: Serial Binary Adder

4.2.5 Parity Generator

When a long string is transmitted over a communication channel, it is possible for some of the symbols to get corrupted. For a binary string, bits may get flipped, i.e., a 0 becomes a 1 and a 1 becomes a 0. There are many sophisticated ways for the receiver to detect such errors and request retransmissions of the relevant portions of the string. I will sketch a relatively simple technique to achieve this.

First, the sender breaks up the string into blocks of equal length. Below I take the block length to be 3, though blocks are much longer in practice . Consider the following input string where spaces separate the blocks.

011 100 010 111

Next, the sender appends a bit at the end of each block so that each 4-bit block has an even number of 1s. This additional bit is called a *parity* bit, and each block is said to have *even parity*. The input string shown above becomes, after addition of parity bits,

0110 1001 0101 1111

This string is transmitted. Suppose two bits are flipped during transmission, as shown below; the flipped bits are underlined.

0110 100$\underline{0}$ 0101 $\underline{0}$111

Note that the flipped bit could be a parity bit or one of the original ones. Now each erroneous block has odd parity, and the receiver can identify all such blocks. It then asks for retransmission of those blocks.

If two bits (or any even number) of bits of a block get flipped, the receiver cannot detect the error. In practice, the blocks are much longer (than 3, shown here) and many additional bits are used for error detection.

The logic at the receiver can be depicted by a finite state acceptor, see Figure 4.12. Here, a block is accepted iff it has even parity. The receiver will ask for retransmission of a block if it enters a reject state for that block (this is not part of the diagram).

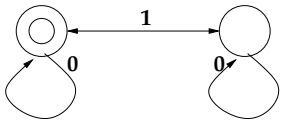


Figure 4.12: Checking the parity of a block of arbitrary length

The sender is a finite state transducer that inserts a bit after every three input bits; see figure 4.13. The start state is 0. The states have the following meanings: in a state numbered $2i$, $0 \leq i \leq 3$, the machine has seen i bits of input of the current block (all blocks are 3 bits long) and the current block parity is even; in state $2i-1$, $1 \leq i \leq 3$, the machine has seen i bits of input of the current block and the current block parity is odd. From states 5 and 6, the machine outputs a parity bit (1 and 0, respectively) without reading an input bit.

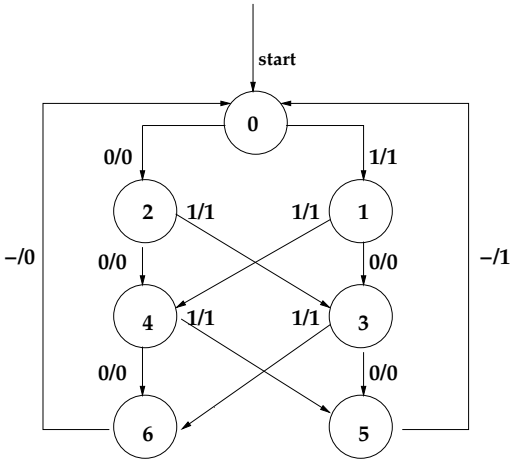


Figure 4.13: Append parity bit to get even parity; block length is 3

0	0	1	0	0	0	1	1	0	0	1	1	0	0	1
Y	*Y*	*Y*	*Y*	*Y*	*Y*	*N*	*Y*	*N*	*Y*	*Y*	*Y*	*N*	*Y*	*Y*

Table 4.5: Checking for valid blocks

Exercise 30

A binary string is *valid* if all blocks of 0s are of even length and all blocks of 1s are of odd length. Design a machine that reads a string and outputs a Y or N for each bit. It outputs N if the current bit ends a block (a block is ended by a bit that differs from the bits in that block) and that block is not valid; otherwise the output is Y. See Table 4.5 for an example. □

4.3 Specifying Control Logic Using Finite State Machines

4.3.1 The Game of Simon

A game that tests your memory —called *Simon*— was popular during the 80s. This is an electronic device that has a number of keys. Each key lights up on being pressed or on receiving an internal pulse.

The game is played as follows. The device lights up a random sequence of keys; call this sequence a *challenge*, and the player is expected to press the same sequence of keys. If the player's response matches the challenge, the device buzzes happily, otherwise sadly. Following a successful response, the device poses a longer challenge. The challenge for which the player loses (the player's response differs from the challenge) is a measure of the memory capability of the player.

We will represent the device by a finite state machine, ignoring the lights and buzzings. Also, we simplify the problem by having 2 keys, marked 0 and 1. Suppose the challenge is a 2-bit sequence (generated randomly within the machine). Figure 4.14 shows a finite state machine that accepts 4 bits of input (2 from the device and 2 from the player) and enters an accepting state only if the first two bits match the last two.

Exercise 31

The device expects the player to press the keys within 30 seconds. If no key is pressed in this time interval, the machine transits to the initial state (and rejects the response). Assume that 30 seconds after the last key press the device receives the symbol p (for pulse) from an internal clock. Modify the machine in Figure 4.14 to take care of this additional symbol. You may assume that p is never received during the input of the first 2 bits. □

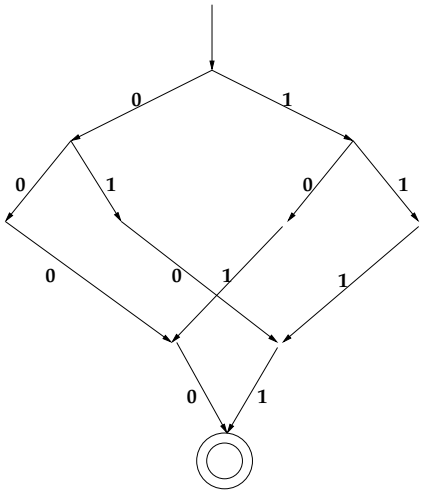


Figure 4.14: A Simplified game of Simon

Remark Finite state machines are used in many applications where the passage of time or exceeding a threshold level for temperature, pressure, humidity, carbon-monoxide, or similar analog measures, causes a state change. A sensor converts the analog signals to digital signals which are then processed by a finite state machine. A certain luxury car has rain sensors mounted in its windshield that detect rain and turn on the wipers. (Be careful when you go to a car wash with this car.) □

4.3.2 Soda Machine

A soda machine interacts with a user to deliver a product. The user provides the input string to the machine by pushing certain buttons and depositing some coins. The machine dispenses the appropriate product provided adequate money has been deposited. Additionally, it may return some change and display warning messages.

We consider a simplified soda machine that dispenses two products, A and B. A costs 15¢ and B 20¢. The machine accepts only nickels and dimes. It operates according to the following rules.

1. If the user presses the appropriate button —a for A and b for B— after depositing at least the correct amount —15¢ for A and 20¢ for B— the machine dispenses the item and returns change, if any, in nickels.

2. If the user inserts additional coins after depositing 20¢ or more, the last coin is returned.

3. If the user asks for an item before depositing the appropriate amount, a warning light flashes for 2 seconds.

4. The user may cancel the transaction at any time. The deposit, if any, is returned in nickels.

The first step in solving the problem is to decide on the input and output alphabets. I propose the following input alphabet:

$\{n,\ d,\ a,\ b,\ c\}$.

Insertion of a nickel (resp., dime) is represented by n (resp., d), pressing the buttons for A (resp., B) is represented by a (resp., b), and pressing the cancellation button is represented by c.

The output alphabet of the machine is

$\{n,\ d,\ A,\ B,\ w\}$.

Returning a nickel (resp., dime) is represented by n (resp., d). A string like nnn represents the return of 3 nickels. Dispensing A (resp., B) is represented by A (resp., B). Flashing the warning light is represented by w.

The machine shown in Figure 4.15 has its states named after the multiples of 5, denoting the total deposit at any point. No other deposit amount is possible, no other number lower than 25 is divisible by 5 (a nickel's value) and no number higher than 25 will be accepted by the machine. (Why do we have a state 25 when the product prices do not exceed 20?) The initial state is 0. In Figure 4.15, all transitions of the form $c/nnn\ldots$ are directed to state 0.

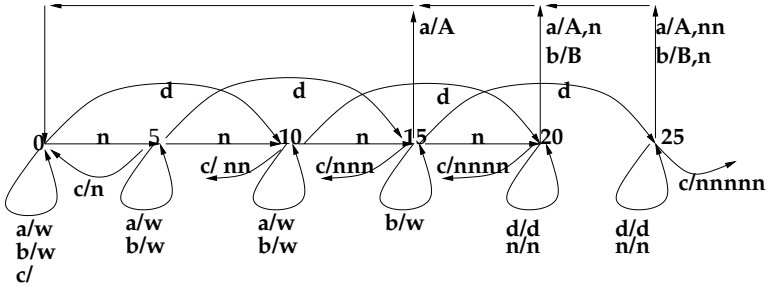


Figure 4.15: Soda Machine; transitions $c/nnn\ldots$ go to state 0

Exercise 32
Design a soda machine that dispenses three products costing 35¢, 55¢ and 75¢. It operates in the same way as the machine described here. □

4.4 Regular Expressions

We have seen so far that a finite state machine is a convenient way of defining certain patterns (but not all). We will study another way, *regular expressions*, of

defining patterns that is exactly as powerful as finite state machines: the same set of patterns can be defined by finite state machines and regular expressions.

Suppose we want to search a file for all occurrences of *integer*; from page 66, an integer is either 0 or a non-zero digit followed by any number of digits. We can define the pattern by a finite state machine. We can also write a definition using a regular expression:

$$0 \mid (1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^*.$$

4.4.1 What is a Regular Expression?

A regular expression is like an arithmetic expression. An arithmetic expression, such as $3 * (x + 5)$, has operands 3, x, 5 and operators $+$ and $*$. For a regular expression, we have an associated *alphabet* that plays the role of constants, like 3 and 5. A regular expression may have operands (like x in the arithmetic expression) that are names of other regular expressions. We have three operators: concatenation (denoted by a period or simple juxtaposition), union or alternation (denoted by $|$) and closure (denoted by *). The first two operators are binary infix operators like $+$ and $*$; the last one is a unary operator, like unary minus, which is written after its operand. More formally, a regular expression defines a set of strings, and it has one of the following forms:

the symbol ϕ, denoting an empty set of strings, or
the symbol ϵ, denoting a set with an empty string, or
a symbol of the alphabet,
  denoting a set with only one string which is that symbol, or
pq, where p and q are regular expressions,
  denoting a set of strings obtained by *concatenation*
  of strings from p with those of q, or
$p \mid q$, where p and q are regular expressions,
  denoting the *union* of the sets corresponding to p and q, or
p^*, where p is a regular expression,
  denoting the *closure* of
  (zero or more concatenations of the strings in) the set corresponding to p.

Binding Power The operators in order of increasing binding power are: alternation, concatenation and closure. So, $\alpha\beta^* \mid \alpha^*\beta$ is $(\alpha(\beta^*)) \mid ((\alpha^*)\beta)$.

Examples of Regular Expressions Here are some regular expressions over the alphabet $\{\alpha, \beta, \gamma\}$.

ϵ, ϕ, α, β, γ
$\epsilon\alpha$, $\alpha\beta$, $\alpha\phi$, $\epsilon\phi\epsilon\phi$
$\alpha\beta \mid \alpha\alpha\beta\epsilon \mid \alpha \mid \epsilon$
$(\alpha\beta)^*(\alpha\alpha\beta\epsilon)^*(\alpha \mid \epsilon \mid \alpha\beta)^*$
$(\alpha(\alpha\beta)^* \mid (\beta\alpha)^*\beta)^*\alpha\beta \mid \alpha\gamma^*\gamma\alpha^*$ □

Each regular expression stands for a set of strings.

Name	Regular expression	Strings
$p =$	$\alpha \mid \alpha\beta \mid \alpha\alpha\beta$	$\{\alpha, \alpha\beta, \alpha\alpha\beta\}$
$q =$	$\beta \mid \beta\gamma \mid \beta\beta\gamma$	$\{\beta, \beta\gamma, \beta\beta\gamma\}$
	pq	$\{\alpha\beta, \alpha\beta\gamma, \alpha\beta\beta\gamma,$ $\alpha\beta\beta, \alpha\beta\beta\gamma, \alpha\beta\beta\beta\gamma,$ $\alpha\alpha\beta\beta, \alpha\alpha\beta\beta\gamma, \alpha\alpha\beta\beta\beta\gamma\}$
	$p \mid q$	$\{\alpha, \alpha\beta, \alpha\alpha\beta, \beta, \beta\gamma, \beta\beta\gamma\}$
	p^*	$\{\epsilon,$ $\alpha, \alpha\beta, \alpha\alpha\beta,$ $\alpha\alpha, \alpha\alpha\beta, \alpha\alpha\alpha\beta,$ $\alpha\beta\alpha, \alpha\beta\alpha\beta,$ $\alpha\beta\alpha\alpha\beta, \ldots\}$

Exercise 33

1. With the given alphabet what are the strings in $\epsilon\alpha$, $\alpha\epsilon$, $\phi\alpha$, $\phi\epsilon$, $\epsilon\phi$?
2. What is the set of strings $(\alpha\beta \mid \alpha\alpha\beta)(\beta\alpha \mid \epsilon\alpha\beta\epsilon)$?
3. What is the set of strings $(\alpha \mid \beta)^*$? □

Note on closure One way to think of closure is as follows:

$$p^* \;=\; \epsilon \mid p \mid pp \mid ppp \mid \ldots$$

The right side is not a legal regular expression because it has an infinite number of terms in it. The purpose of closure is to make the right side a regular expression. □

4.4.2 Examples of Regular Expressions

1. $a \mid bc^*d$ is $\{a, bd, bcd, bccd, bcccd, \ldots\}$.
2. All integers are defined by $(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^*$. How would you avoid the "empty" integer?
3. Define an integer to be either a 0 or a nonzero digit followed by any number of digits:
   $0 \mid (1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^*$.
   Show that 3400 is an integer, but 0034 is not.

   The definition of an integer can be simplified by naming certain subexpressions of the regular expression.

   $$\begin{aligned} \mathit{Digit} &= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ \mathit{pDigit} &= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ \mathit{integer} &= 0 \mid (\mathit{pDigit}\ \mathit{Digit}^*) \end{aligned}$$

4. Legal identifiers in Java. Note that a single letter is an identifier.

$$\begin{aligned} &\mathit{Letter} = A \mid B \mid \ldots\ Z \mid a \mid b \mid \ldots\ z \\ &\mathit{Digit} = 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ &\mathit{identifier} = \mathit{Letter}(\mathit{Letter} \mid \mathit{Digit})^* \end{aligned}$$

5. Words in which all the vowels appear in order:

$$(\mathit{Letter}^*)a(\mathit{Letter}^*)e(\mathit{Letter}^*)i(\mathit{Letter}^*)o(\mathit{Letter}^*)u(\mathit{Letter}^*)$$

6. From page 66, a *number* is either an integer or an integer followed by a period followed by any number of digits.

$$\mathit{integer} \mid \mathit{integer}.\mathit{Digit}^*$$

7. An *increasing integer* is a nonempty integer whose digits are strictly increasing. Let us define int_i, for $0 \leq i \leq 9$, to be an increasing integer whose first digit is greater than or equal to i. Then, an increasing integer is

$$\mathit{IncInt} = int_0$$

Next, let us define int_i, $0 \leq i \leq 9$. It is easier to start with the highest index, 9, and work downwards.

$$\begin{aligned} int_9 &= 9 \\ int_8 &= int_9 \mid 8\ int_9 = (\epsilon \mid 8)int_9 \\ int_7 &= int_8 \mid 7\ int_9 = (\epsilon \mid 7)int_8 \\ int_i &= int_{i+1} \mid i\ int_{i+1} = (\epsilon \mid i)int_{i+1}, \text{ for } 0 \leq i < 9 \end{aligned}$$

4.4.3 Solving Regular Expression Equations

We find it convenient to write a long definition, such as that of *IncInt*, by using a number of sub-definitions, such as int_9 through int_0. It is often required to eliminate all such variables from a regular expression and get a (long) expression in which only the symbols of the alphabet appear. We can do it easily for a term like int_7 that is defined to be $7(\epsilon \mid int_8 \mid int_9)$; replace int_8 by its definition to get $7(\epsilon \mid (8 \mid 8int_9) \mid int_9)$, and int_9 by its definition to get $7(\epsilon \mid (8 \mid 89) \mid 9)$, i.e., $7 \mid (78 \mid 789) \mid 79$. However, in many cases the equations are recursive, so this trick will not work. For instance, let *pe* and *po* be the binary strings that have even number of 1s and odd number of 1s, respectively. Then,

$$\begin{aligned} p &= \epsilon \mid 0^*1q \\ q &= 0^*1p \end{aligned}$$

Replace q in the definition of p to get

$$p = \epsilon \mid 0^*10^*1p$$

This is a recursive equation. We see that $p = \epsilon \mid \alpha p$ where α is some string that does not name p ($\alpha = 0^*10^*1$). Then $p = \alpha^*$, that is, $p = (0^*10^*1)^*$. Hence, $q = 0^*1(0^*10^*1)^*$.

A more elaborate example It is required to define a binary string that is a multiple of 3 considered as a number. Thus, 000 and 011 are acceptable strings, but 010 is not. Let b_i, $0 \leq i \leq 2$, be a binary string that leaves a remainder of i after division by 3. We have:

$$b_0 = \epsilon \mid b_0 0 \mid b_1 1 \quad (1)$$
$$b_1 = b_0 1 \mid b_2 0 \quad (2)$$
$$b_2 = b_1 0 \mid b_2 1 \quad (3)$$

These equations can be understood by answering the following questions: on division by 3 if p leaves a remainder of i, $0 \leq i \leq 2$, then what are the remainders left by $p0$ and $p1$? Note that letting $v(p)$ denote the value of string p as an integer, $v(p0) = 2v(p)$ and $v(p1) = 2v(p) + 1$.

We solve these equations to create a regular expression for b_0. First, we have the following observation. For variable z and regular expressions a and y:

$$z = a \mid zy$$

has the solution $z = ay^*$. We prove the validity of this observation by substituting ay^* for z in the equation, and showing that that the two sides are equal.

$$\begin{array}{ll} & a \mid zy \\ = & \{\text{replace } z \text{ by } ay^*\} \\ & a \mid ay^*y \\ = & \{\text{apply (7) in Section 4.4.4}\} \\ & a(\epsilon \mid y^*y) \\ = & \{y^* = \epsilon \mid y^*y, \text{ see (9) in Section 4.4.4}\} \\ & ay^* \\ = & \{\text{replace } ay^* \text{ by } z\} \\ & z \end{array}$$

Apply this observation to (3) with z, a, y set to b_2, $b_1 0$, 1 to get

$$b_2 = b_1 01^*$$

In the RHS of (2), replace b_2 by $b_1 01^*$:

$$b_1 = b_0 1 \mid b_1 01^* 0$$

Apply the observation on this equation with z, a, y set to b_1, $b_0 1$, 01^*0.

$$b_1 = b_0 1(01^*0)^*$$

Replace b_1 in the RHS of (1) by the RHS above.

$$\begin{array}{ll} b_0 = & \epsilon \mid b_0 0 \mid b_0 1(01^*0)^*1, \text{ or} \\ = & \epsilon \mid b_0(0 \mid 1(01^*0)^*1) \end{array}$$

Apply the observation with z, a, y set to b_0, ϵ, $(0 \mid 1(01^*0)^*1)$.

$$\begin{array}{ll} b_0 = & \epsilon\ (0 \mid 1(01^*0)^*1)^*, \text{ or} \\ = & (0 \mid 1(01^*0)^*1)^* \end{array}$$

4.4.4 Algebraic Properties of Regular Expressions

1. Identity for Union $(\phi \mid R) = R$, $(R \mid \phi) = R$
2. Identity for Concatenation $(\epsilon R) = R$, $(R\epsilon) = R$
3. $(\phi R) = \phi$, $(R\phi) = \phi$
4. Commutativity of Union $(R \mid S) = (S \mid R)$
5. Associativity of Union $((R \mid S) \mid T) = (R \mid (S \mid T))$
6. Associativity of Concatenation $((RS)T) = (R(ST))$
7. Distributivity of Concatenation over Union
   $(R(S \mid T)) = (RS \mid RT)$
   $((S \mid T)R) = (SR \mid TR)$
8. Idempotence of Union $(R \mid R) = R$
9. Closure
   $\phi^* = \epsilon$
   $RR^* = R^*R$
   $R^* = (\epsilon \mid RR^*)$

Exercise 34
Write regular expressions for the following sets of binary strings.

1. Strings whose numerical values are even.
2. Strings whose numerical values are non-zero.
3. Strings that have at least one 0 and at most one 1.
4. Strings in which the 1s appear contiguously.
5. Strings in which every substring of 1s is of even length. □

Exercise 35
Define the language over the alphabet $\{0,\ 1,\ 2\}$ in which consecutive symbols are different. □

Exercise 36
What are the languages defined by

1. $(0^*1^*)^*$
2. $(0^* \mid 1^*)^*$
3. ϵ^*
4. $(0^*)^*$
5. $(\epsilon \mid 0^*)^*$ □

4.4.5 From Regular Expressions to Machines

Regular expressions and finite state machines are equivalent: for each regular expression R there exists a finite state machine F such that the set of strings in R is the set of strings accepted by F. The converse also holds. I will not prove this result, but will instead give an informal argument.

First, let us construct a machine to recognize the single symbol b. The machine has a start state S, an accepting state F and a rejecting state G. There is a transition from S to F labeled b, a transition from S to G labeled with all other symbols and a transition from F to G labeled with all symbols. The machine remains in G forever (i.e., for all symbols the machine transits from G to G), see Figure 4.16. In this figure, *Alph* stands for the alphabet.

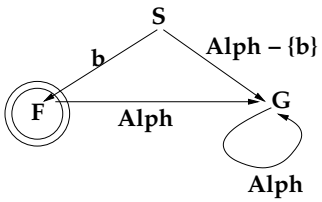


Figure 4.16: Machine that accepts b

Henceforth, we will not show a permanently rejecting state, such as G, explicitly. If you see no transition from a state with a given symbol, assume that it transits to a permanently rejecting state. So, we will simplify Figure 4.16 to Figure 4.17.

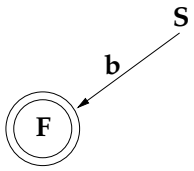


Figure 4.17: Machine that accepts b, simplified

How do we construct a machine to recognize concatenation? Suppose we have a machine that accepts a regular expression p and another machine that accepts q. Suppose p's machine has a single accepting state. Then we can merge the two machines by identifying the accepting state of the first machine with the start state of the second. We will see an example of this below. You may think about how to generalize this construction when the first machine has several accepting states.

Next, let us consider closure. Suppose we have to accept c^*. The machine in Figure 4.18 does the job.

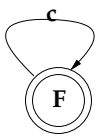


Figure 4.18: Machine that accepts c^*

Now, let us put some of these constructions together and build a machine to recognize bc^*. The machine is shown in Figure 4.19.

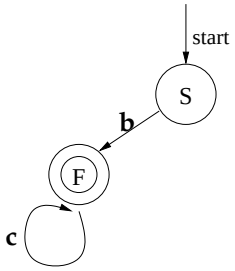


Figure 4.19: Machine that accepts bc^*

You can see that it is a concatenation of a machine that accepts b and one that accepts c^*. Next, let us construct a machine that accepts $bc^* \mid cb^*$. Clearly, we can build machines for both bc^* and cb^* separately. Building their union is easy, because bc^* and cb^* start out with different symbols, so we can decide which machine should scan the string, as shown in Figure 4.20.

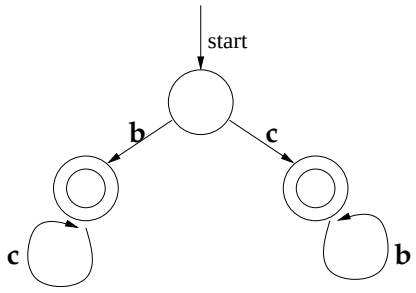


Figure 4.20: Machine that accepts $bc^* \mid cb^*$

Exercise 37

Construct a machine to accept $bc^* \mid bd^*$. □

4.5 Regular Expressions in Practice; from GNU Emacs

The material in this section is taken from the online GNU Emacs manual[1].

Regular expressions have a syntax in which a few characters are special constructs and the rest are "ordinary". An ordinary character is a simple regular expression which matches that same character and nothing else. The special characters are '$', '^', '.', '*', '+', '?', '[', ']' and '\'. Any other character appearing in a regular expression is ordinary, unless a '\' precedes it.

For example, 'f' is not a special character, so it is ordinary, and therefore 'f' is a regular expression that matches the string 'f' and no other string. (It does *not* match the string 'ff'.) Likewise, 'o' is a regular expression that matches only 'o'. (When case distinctions are being ignored, these regexps also match 'F' and 'O', but we consider this a generalization of "the same string", rather than an exception.)

Any two regular expressions A and B can be concatenated. The result is a regular expression which matches a string if A matches some amount of the beginning of that string and B matches the rest of the string.

As a simple example, we can concatenate the regular expressions 'f' and 'o' to get the regular expression 'fo', which matches only the string 'fo'. Still trivial. To do something nontrivial, you need to use one of the special characters. Here is a list of them.

'. (Period)' is a special character that matches any single character except a newline. Using concatenation, we can make regular expressions like 'a.b' which matches any three-character string which begins with 'a' and ends with 'b'.

'*' is not a construct by itself; it is a postfix operator, which means to match the preceding regular expression repetitively as many times as possible. Thus, 'o*' matches any number of 'o's (including no 'o's).

'*' always applies to the *smallest* possible preceding expression. Thus, 'fo*' has a repeating 'o', not a repeating 'fo'. It matches 'f', 'fo', 'foo', and so on.

The matcher processes a '*' construct by matching, immediately, as many repetitions as can be found. Then it continues with the rest of the pattern. If that fails, backtracking occurs, discarding some of the matches of the '*'-modified construct in case that makes it possible to match the rest of the pattern. For example, matching 'ca*ar' against the string 'caaar', the 'a*' first tries to match all three 'a's; but the rest of the pattern is 'ar' and there is only 'r' left to match, so this try fails. The next alternative is for 'a*' to match only two 'a's. With this choice, the rest of the regexp matches successfully.

'+' is a postfix character, similar to '*' except that it must match the preceding expression at least once. So, for example, 'ca+r' matches the strings 'car' and 'caaaar' but not the string 'cr', whereas 'ca*r' matches all three strings.

---

[1] Copyright (C) 1989,1991 Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA

‘?’ is a postfix character, similar to ‘*’ except that it can match the preceding expression either once or not at all. For example, ‘ca?r’ matches ‘car’ or ‘cr’; nothing else.

‘[...]’ is a "character set", which begins with ‘[’ and is terminated by a ‘]’. In the simplest case, the characters between the two brackets are what this set can match.

Thus, ‘[ad]’ matches either one ‘a’ or one ‘d’, and ‘[ad]*’ matches any string composed of just ‘a’s and ‘d’s (including the empty string), from which it follows that ‘c[ad]*r’ matches ‘cr’, ‘car’, ‘cdr’, ‘caddaar’, etc.

You can also include character ranges a character set, by writing two characters with a ‘-’ between them. Thus, ‘[a-z]’ matches any lower-case letter. Ranges may be intermixed freely with individual characters, as in ‘[a-z$%.]’, which matches any lower case letter or ‘$’, ‘%’ or period.

Note that the usual special characters are not special any more inside a character set. A completely different set of special characters exists inside character sets: ‘]’, ‘-’ and ‘^’.

To include a ‘]’ in a character set, you must make it the first character. For example, ‘[]a]’ matches ‘]’ or ‘a’. To include a ‘-’, write ‘-’ at the beginning or end of a range. To include ‘^’, make it other than the first character in the set.

‘[^ ...]’ ‘[^’ begins a "complemented character set", which matches any character except the ones specified. Thus, ‘[^a-z0-9A-Z]’ matches all characters *except* letters and digits.

‘^’ is not special in a character set unless it is the first character. The character following the ‘^’ is treated as if it were first (‘-’ and ‘]’ are not special there).

A complemented character set can match a newline, unless newline is mentioned as one of the characters not to match. This is in contrast to the handling of regexps in programs such as ‘grep’.

‘^’ is a special character that matches the empty string, but only at the beginning of a line in the text being matched. Otherwise it fails to match anything. Thus, ‘^foo’ matches a ‘foo’ which occurs at the beginning of a line.

‘$’ is similar to ‘^’ but matches only at the end of a line. Thus, ‘xx*$’ matches a string of one ‘x’ or more at the end of a line.

‘\’ has two functions: it quotes the special characters (including ‘\’), and it introduces additional special constructs.

Because ‘\’ quotes special characters, ‘\$’ is a regular expression which matches only ‘$’, and ‘\[’ is a regular expression which matches only ‘[’, etc.

For the most part, ‘\’ followed by any character matches only that character. However, there are several exceptions: two-character sequences starting with ‘\’ which have special meanings. The second character in the sequence is always an ordinary character on their own. Here is a table of ‘\’ constructs.

‘\|’ specifies an alternative. Two regular expressions A and B with ‘\|’ in between form an expression that matches anything that either A or B matches.

Thus, ‘foo\|bar’ matches either ‘foo’ or ‘bar’ but no other string.

‘\|’ applies to the largest possible surrounding expressions. Only a surrounding ‘\(... \)’ grouping can limit the scope of ‘\|’.

Full backtracking capability exists to handle multiple uses of '`\|`'.

'`\( ... \)`' is a grouping construct that serves three purposes:

1. To enclose a set of '`\|`' alternatives for other operations. Thus, '`\(foo\|bar\)x`' matches either 'foox' or 'barx'.

2. To enclose a complicated expression for the postfix operators '*', '+' and '?' to operate on. Thus, '`ba\(na\)*`' matches 'bananana', etc., with any (zero or more) number of 'na' strings.

3. To mark a matched substring for future reference.

This last application is not a consequence of the idea of a parenthetical grouping; it is a separate feature which is assigned as a second meaning to the same '`\( ... \)`' construct. In practice there is no conflict between the two meanings. Here is an explanation of this feature:

'`\D`' after the end of a '`\( ... \)`' construct, the matcher remembers the beginning and end of the text matched by that construct. Then, later on in the regular expression, you can use '\' followed by the digit D to mean "match the same text matched the Dth time by the '`\( ... \)`' construct."

The strings matching the first nine '`\( ... \)`' constructs appearing in a regular expression are assigned numbers 1 through 9 in order that the open-parentheses appear in the regular expression. '`\1`' through '`\9`' refer to the text previously matched by the corresponding '`\( ... \)`' construct.

For example, '`\(.*\)\1`' matches any newline-free string that is composed of two identical halves. The '`\(.*\)`' matches the first half, which may be anything, but the '`\1`' that follows must match the same exact text.

If a particular '`\( ... \)`' construct matches more than once (which can easily happen if it is followed by '*'), only the last match is recorded.

'`\‘`' matches the empty string, provided it is at the beginning of the buffer.

'`\’`' matches the empty string, provided it is at the end of the buffer.

'`\b`' matches the empty string, provided it is at the beginning or end of a word. Thus, '`\bfoo\b`' matches any occurrence of 'foo' as a separate word. '`\bballs?\b`' matches 'ball' or 'balls' as a separate word.

'`\B`' matches the empty string, provided it is *not* at the beginning or end of a word.

'`\<`' matches the empty string, provided it is at the beginning of a word.

'`\>`' matches the empty string, provided it is at the end of a word.

'`\w`' matches any word-constituent character. The syntax table determines which characters these are.

'`\W`' matches any character that is not a word-constituent.

'`\sC`' matches any character whose syntax is C. Here C is a character which represents a syntax code: thus, 'w' for word constituent, '(' for open-parenthesis, etc. Represent a character of whitespace (which can be a newline) by either '-' or a space character.

'`\SC`' matches any character whose syntax is not C.

The constructs that pertain to words and syntax are controlled by the setting of the syntax table.

Here is a complicated regexp, used by Emacs to recognize the end of a sentence together with any whitespace that follows. It is given in Lisp syntax to

enable you to distinguish the spaces from the tab characters. In Lisp syntax, the string constant begins and ends with a double-quote. '`\"`' stands for a double-quote as part of the regexp, '`\\`' for a backslash as part of the regexp, '`\t`' for a tab and '`\n`' for a newline.

```
"[.?!][]\"')]*\\($\\|\t\\|  \\)[ \t\n]*"
```

This contains four parts in succession: a character set matching period, '?', or '!'; a character set matching close-brackets, quotes, or parentheses, repeated any number of times; an alternative in backslash-parentheses that matches end-of-line, a tab, or two spaces; and a character set matching whitespace characters, repeated any number of times.

To enter the same regexp interactively, you would type TAB to enter a tab, and 'C-q C-j' to enter a newline. You would also type single slashes as themselves, instead of doubling them for Lisp syntax.

Chapter 5

Recursion and Induction

5.1 Introduction

In this set of lectures, I will talk about *recursive programming*, a programming technique you have seen before. I will introduce a style of programming, called *Functional Programming*, that is especially suited for describing recursion in computation and data structure. Functional programs are often significantly more compact, and easier to design and understand, than their imperative counterparts. I will show why induction is an essential tool in designing functional programs.

Haskell I will use a functional programming language, called `Haskell`. What follows is a very very small subset of Haskell; you should consult the references given at the end of this document for further details. A very good source is Thompson [48] which covers this material with careful attention to problems that students typically face. Another very good source is Richards [40], whose lecture slides are available online. The Haskell manual is available online at [18]; you should consult it as a reference, particularly its Prelude (haskell98-report/standard-prelude.html) for definitions of many built-in functions. However, the manual is not a good source for learning programming. I would recommend the book by Bird [6], which teaches a good deal of programming methodology. Unfortunately, the book does not quite use Haskell syntax, but a syntax quite close to it. Another good source is *A Gentle Introduction to Haskell* [19] which covers all the material taught here, and more, in its first 20 pages.

The Haskell compiler is installed on all Sun and Linux machines in this department. To enter an interactive session for Haskell, type

```
hugs
```

The machine responds with something like

```
Haskell 98 mode: Restart with command line option -98 to enable extensions
```

```
Reading file "/lusr/share/hugs/lib/Prelude.hs":

Hugs session for:
/lusr/share/hugs/lib/Prelude.hs
Type :? for help
Prelude>
```

At this point, whenever it displays `zzz>` for some `zzz`, the machine is waiting for some response from you. You may type an expression and have its value displayed on your terminal, as follows.

```
Prelude> 3+4
7
Prelude> 2^15
32768
```

5.2 Primitive Data Types

Haskell has a number of built-in data types; we will use only integer (called *Int*), boolean (called *Bool*), character (called *Char*) and string (called *String*) types[1].

Integer You can use the traditional integer constants and the usual operators for addition (`+`), subtraction (`-`) and multiplication (`*`). Unary minus sign is the usual `-`, but enclose a negative number within parentheses, as in `(-2)`; I will tell you why later. Division over integers is written in infix as `` `div` `` and it returns only the integer part of the quotient; thus, `5 `div` 3` is `1` and `(-5) `div` 3` is `-2` (`` ` `` is the backquote symbol, usually it is the leftmost key in the top row of your keyboard). Exponentiation is the infix operator `^` so that `2^15` is `32768`. The remainder after division is given by the infix operator `` `rem` `` and the infix operator `` `mod` `` is for modulo; `x `mod` y` returns a value between 0 and $y-1$, for positive y. Thus, `(-2) `rem` 3` is -2 and `(-2) `mod` 3` is 1.

Two other useful functions are `even` and `odd`, which return the appropriate boolean results about their integer arguments. Functions `max` and `min` take two arguments each and return the maximum and the minimum values, respectively. It is possible to write a function name followed by its arguments without any parentheses, as shown below; parentheses are needed only to enforce an evaluation order.

```
Prelude> max 2 5
5
```

The arithmetic relations are `< <= == /= > >=`. Each of these is a binary operator that returns a boolean result, `True` or `False`. Note that equality

[1] Haskell supports two kinds of integers, `Integer` and `Int` data types. We will use only `Int`.

operator is written as `==` and inequality as `/=`. Unlike in C++, `3 + (5 >= 2)` is not a valid expression; Haskell does not specify how to add an integer to a boolean.

Boolean There are the two boolean constants, written as `True` and `False`. The boolean operators are:

```
not -- for negation
&&  -- for and
||  -- for or
==  -- for equivalence
/=  -- for inequivalence (also called "exclusive or")
```

Here is a short session with hugs.

```
Prelude> (3 > 5) || (5 > 3)
True
Prelude> (3 > 3) || (3 > 3)
False
Prelude> (2 `mod` (-3)) == ((-2) `mod` 3)
False
Prelude> even 3 || odd 3
True
```

Character and String A character is enclosed within single quotes and a string is enclosed within double quotes.

```
Prelude> 'a'
'a'
Prelude> "a b c"
"a b c"
Prelude> "a, b, c"
"a, b, c"
```

You can compare characters using arithmetic relations; the letters (characters in the Roman alphabet) are ordered in the usual fashion with the uppercase letters *smaller* than the corresponding lowercase letters. The expected ordering applies to the digits as well.

```
Prelude> 'a' < 'b'
True
Prelude> 'A' < 'a'
True
Prelude> '3' < '5'
True
```

There are two functions defined on characters, `ord` and `chr`. Function `ord(c)` returns the value of the character `c` in the internal coding table; it is a number between 0 and 255. Function `chr` converts a number between 0 and 255 to the corresponding character. Therefore, `chr(ord(c))=c`, for all characters `c`, and `ord(chr(i))=i`, for all `i`, $0 \leq$ `i` $<$ `256`. Note that all digits in the order `'0'` through `'9'`, all lowercase letters 'a' through 'z' and uppercase letters 'A' through 'Z' are contiguous in the table. The uppercase letters have smaller *ord* values than the lowercase ones.

```
Prelude> ord('a')
97
Prelude> chr(97)
'a'
Prelude> ord(chr(103))
103
Prelude> chr(255)
'\255'
Prelude> (ord '9')- (ord '0')
9
Prelude> (ord 'a')- (ord 'A')
32
```

A string is a *list* of characters; all the rules about lists, described later, apply to strings.

5.3 Writing Function Definitions

We cannot compute much by working with constants alone. We need to be able to define functions. The functions cannot be defined by interactive input. We need to keep a file in which we list all the definitions and load that file.

5.3.1 Loading Program Files

Typically, Haskell program files have the suffix `hs`. I load a file `Utilities.hs`, which I have stored in a directory called `haskell.dir`, as follows.

```
Prelude> :l haskell.dir/Utilities.hs
Reading file "haskell.dir/Utilities.hs":

Hugs session for:
/lusr/share/hugs/lib/Prelude.hs
haskell.dir/Utilities.hs
```

I have a program in that file to sort a list of numbers. So, now I may write

```
Utilities> sort[7, 5, 1, 9]
[1,5,7,9]
```

Let me create a file `337.hs` in which I will load all the definitions in this note. Each time you change a file, by adding or modifying definitions, you have to reload the file into hugs (a quick way to do this is to use the hugs command `:r` which reloads the last loaded file).

5.3.2 Comments

Any string following `--` in a line is considered a comment. So, you may write in a command line:

```
Prelude> 2^15   -- This is 2 raised to 15
```

or, in the text of a program

```
-- I am now going to write a function called "power."
-- The function is defined as follows:
--  It has 2 arguments and it returns
--  the first argument raised to the second argument.
```

There is a different way to write comments in a program that works better for longer comments, like the four lines I have written above: you can enclose a region within `{-` and `-}` to make the region a comment.

```
{- I am now going to write a function called "power."
   The function is defined as follows:
    It has 2 arguments and it returns
    the first argument raised to the second argument. -}
```

I prefer to put the end of the comment symbol, `-}`, in a line by itself.

5.3.3 Examples of Function Definitions

In its simplest form, a function is defined by: (1) writing the function name, (2) followed by its arguments, (3) then a "=", (4) followed by the body of the function definition.

Here are some simple function definitions. Note that I do not put any parentheses around the arguments, they are simply written in order and parentheses are put only to avoid ambiguity. We will discuss this matter in some detail later, in Section 5.4.2 (page 99).

Note: Parameters and arguments I will use these two terms synonymously. □

```
inc x = x+1                             -- increment x
imply p q = not p || q                  -- boolean implication
digit c =  ('0' <= c) && (c <= '9')     -- is c a digit?
```

We test some of our definitions now.

```
Main> :l 337.hs
Reading file "337.hs":

Hugs session for:
/lusr/share/hugs/lib/Prelude.hs
337.hs
Main> inc 5
6
Main> imply True False
False
Main> digit '6'
True
Main> digit 'a'
False
Main> digit(chr(inc (ord '8')))
True
Main> digit(chr(inc (ord '9')))
False
```

We can use other function names in a function definition.

We can define variables in the same way we define functions; a variable is a function without arguments. For example,

```
offset  = (ord 'a') - (ord 'A')
```

Unlike a variable in C++, this variable's value does not change during the program execution; we are really giving a name to a constant expression so that we can use this name for easy reference later.

Exercise 38

1. Write a function to test if its argument is a lowercase letter; write another to test if its argument is an uppercase letter.

2. Write a function to test if its argument, an integer, is divisible by 6.

3. Write a function whose argument is an uppercase letter and whose value is the corresponding lowercase letter.

4. Define a function whose argument is a digit, 0 through 9, and whose value is the corresponding character `'0'` through `'9'`.

5. Define a function `max3` whose arguments are three integers and whose value is their maximum. □

5.3.4 Conditionals

In traditional imperative programming, we use `if-then-else` to test some condition (i.e., a predicate) and perform calculations based on the test. Haskell also provides an `if-then-else` construct, but it is often more convenient to use a *conditional equation*, as shown below. The following function computes the absolute value of its integer argument.

```
absolute x
 | x >= 0 = x
 | x < 0  = -x
```

The entire definition is a *conditional equation* and it consists of two *clauses*. A clause starts with a bar (|), followed by a predicate (called a *guard*), an equals sign (=) and the expression denoting the function value for this case. The guards are evaluated in the order in which they are written (from top to bottom), and for the first guard that is true, its corresponding expression is evaluated. So, if you put `x <= 0` as the second guard in the example above, it will work too, but when `x = 0` the expression in the first equation will be evaluated and the result returned.

You can write `otherwise` for a guard, denoting a predicate that holds when none of the other guards hold. An `otherwise` guard appears only in the last equation. The same effect is achieved by writing `True` for the guard in the last equation. If no guard is `True` in a conditional equation, you will get a run-tme error message.

Given below is a function that converts the argument letter from upper to lowercase, or lower to uppercase, as is appropriate. Assume that we have already written two functions, `uplow` (to convert from upper to lowercase), `lowup` (to convert from lower to uppercase), and a function `upper` whose value is `True` iff its argument is an uppercase letter.

```
chCase c                  -- change case
 | upper c   = uplow c
 | otherwise = lowup c
```

The test with hugs gives:

```
Main> chCase 'q'
'Q'
Main> chCase 'Q'
'q'
```

Exercise 39

1. Define a function that returns `-1`, `0` or `+1`, depending on whether the argument is negative, zero or positive.

2. Define a function that takes three integer arguments, p, q and r. If these arguments are the lengths of the sides of a triangle, the function value is `True`; otherwise, it is `False`. Recall from geometry that every pair of values from p, q and r must sum to a value greater than the third one for these numbers to be the lengths of the sides of a triangle.

```
max3 p q r = max p (max q r)
triangle p q r = (p+q+r) > (2* (max3 p q r))
```

5.4 Lexical Issues

5.4.1 Program Layout

Haskell uses line indentations in the program to delineate the scope of definitions. A definition is ended by a piece of text that is to the left (columnwise) of the start of its definition. Thus,

```
chCase c                                  -- change case
 | upper c   = uplow c
 | otherwise = lowup c
```

and

```
ch1Case c                                  -- change case
 | upper c   = uplow c
   | otherwise          =
 lowup c
```

are fine. But,

```
ch2Case c                                  -- change case
 | upper c   = uplow c
 | otherwise          =
lowup c
```

is not. The line `lowup c` is taken to be the start of another definition. In the last case, you will get an error message like

```
ERROR "337.hs":81 - Syntax error in expression (unexpected ‘;’,
possibly due to bad layout)
```

The semicolon (`;`) plays an important role; it closes off a definition (implicitly, even if you have not used it). That is why you see `unexpected ‘;’` in the error message.

5.4.2 Function Parameters and Binding

Consider the expression

```
f 1+1
```

where `f` is a function of one argument. In normal mathematics, this will be an invalid expression, and if forced, you will interpret it as `f(1+1)`. In Haskell, this is a valid expression and it stands for `(f 1)+1`. I give the binding rules below.

An expression consists of functions, operators and operands as in

```
-2 + sqr 9 + min 2 7 - 3
```

Here the first minus (called unary minus) is a prefix operator, `sqr` is a function of one argument, `+` is a binary operator, `min` is a function of two arguments, and the last minus is a binary infix operator. An operator is a function; a binary operator is typically written between its arguments (in the infix style).

Functions bind more tightly than operators. Function arguments are written immediately following the function name, and the right number of arguments are used up for each function, e.g., one for `sqr` and two for `min`. No parentheses are needed unless your arguments are themselves expressions. So, for a function `g` of two arguments, `g x y z` stands for `(g x y) z`. If you write `g f x y`, it will be interpreted as `(g f x) y`; so, if you have in mind the expression `g(f(x),y)`, write it as `g (f x) y`. But, you can't write `g(f(x),y)`, because `(f(x),y)` will be interpreted as a pair, which is a single item, see Section 5.7 (page 109). Now, `sqr 9 + min 2 7 - 3` is `(sqr 9) + (min 2 7) - 3`. As a good programming practice, do not ever write `f 1+1`; make your intentions clear by using parentheses, as in `(f 1)+1` or `f(1+1)`.

How do we read `sqr 9 + min 2 7` × `max 2 7`? After functions are bound to their arguments, we get `(sqr 9) + (min 2 7)` × `(max 2 7)`. That is, we are left with operators only, and the operators bind according to their binding powers. Since × has higher binding power than +, the expression is read as `(sqr 9) + ((min 2 7)` × `(max 2 7))`.

Operators of equal binding power usually associate to the left; so, `5 - 3 - 2` is `(5 - 3) - 2`, but this is not always true. Operators in Haskell are either (1) associative, so that the order does not matter, (2) left associative, as in binary minus shown above, or (3) right associative, as in `2 ^ 3 ^ 5`, which is `2 ^ (3 ^ 5)`. When in doubt, parenthesize.

In this connection, unary minus, as in `-2`, is particularly problematic. If you would like to apply `inc` to `-2`, don't write

```
inc -2
```

This will be interpreted as `(inc) - (2)`; you will get an error message. Write `inc (-2)`. And `-5 ‘div‘ 3` is `-(5 ‘div‘ 3)` which is`-1`, but `(-5) ‘div‘ 3` is `-2`.

Exercise 40
What is `max 2 3 + min 2 3`? □

Note on Terminology In computing, it is customary to say that a function "takes an argument" and "computes (or returns) a value". A function, being a concept, and not an artifact, cannot "do" anything; it simply has arguments and it has a value for each set of arguments. Yet, the computing terminology is so prevalent that I will use these phrases without apology in these notes.

5.4.3 The `where` Clause

The following function has three arguments, x, y and z, and it determines if $x^2 + y^2 = z^2$.

```
pythagoras x y z = (x*x) + (y*y) == (z*z)
```

The definition would be simpler to read in the following form

```
pythagoras x y z = sqx + sqy == sqz
                   where
                    sqx = x*x
                    sqy = y*y
                    sqz = z*z
```

The `where` construct permits *local definitions*, i.e., defining variables (and functions) within a function definition. The variables `sqx`, `sqy` and `sqz` are undefined outside this definition.

We can do this example by using a local function to define squaring.

```
pythagoras x y z = sq x + sq y == sq z
                   where
                    sq p = p*p
```

5.4.4 Pattern Matching

Previously, we wrote a function like

```
imply p q = not p || q
```

as a single equation. We can also write it in the following form in which there are two equations.

```
imply False q = True
imply True  q = q
```

Observe that the equations use constants in the left side; these constants are called *literal parameters*. During function evaluation with a specific argument —say, `False True`— each of the equations are checked from top to bottom to find the first one where the given arguments match the *pattern* of the equation. For `imply False True`, the pattern given in the first equation matches, with `False` matching `False` and `q` matching `True`.

We can write an even more elaborate definition of `imply`:

```
imply False False = True
imply False True  = True
imply True  False = False
imply True  True  = True
```

The function evaluation is simply a table lookup in this case, proceeding sequentially from top to bottom.

Pattern matching has two important effects: (1) it is a convenient way of doing case discrimination without writing a sphagetti of if-then-else statements, and (2) it *binds* names to formal parameter values, i.e., assigns names to components of the data structure —`q` in the first example— which may be used in the function definition in the right side.

Pattern matching on integers can use simple arithmetic expressions, as shown below in the definition of the *successor* function.

```
suc 0 = 1
suc (n+1) = (suc n)+1
```

Asked to evaluate `suc 3`, the pattern in the second equation is found to match — with `n = 2`— and therefore, evaluation of `(suc 3)` is reduced to the evaluation of `(suc 2) + 1`.

Pattern matching can be applied in elaborate fashions, as we shall see later.

5.5 Recursive Programming

Recursive programming is closely tied to *problem decomposition*. In program design, it is common to divide a problem into a number of subproblems where each subproblem is easier to solve by some measure, and the solutions of the subproblems can be combined to yield a solution to the original problem. A subproblem is easier if its solution is known, or if it is an instance of the original problem, but over a smaller data set. For instance, if you have to sum twenty numbers, you may divide the task into four subtasks of summing five numbers each, and then add the four results. A different decomposition may (1) scan the numbers, putting negative numbers in one subset, discarding zeros and putting positive numbers in another subset, (2) sum the positive and negative subsets individually, and (3) add the two answers. In this case, the first subproblem is different in kind from the original problem.

In recursive programming, typically, a problem is decomposed into subproblems of the same kind, and we apply the same solution procedure to each of the subproblems, further subdividing them. A recursive program has to specify the solutions for the very smallest cases, those which cannot be decomposed any further.

The theoretical justification of recursive programming is *mathematical induction*. In fact, recursion and induction are so closely linked that they are often mentioned in the same breath (see the title of this note); I believe we should have used a single term for this concept.

5.5.1 Computing Powers of 2

Compute $2^{\texttt{n}}$, for $\texttt{n} \geq 0$, using only doubling. As in typical induction, we consider two cases, a base value of `n` and the general case where `n` has larger values. Let us pick the base value of `n` to be `0`; then the function value is `1`. For `n+1` the function value is double the function value of `n`.

```
power2 0     = 1
power2 (n+1) = 2 * (power2 n)
```

How does the computer evaluate a call like `power2 3`? Here is a very rough sketch. The interpreter has an expression to evaluate at any time. It picks an operand (a subexpression) to reduce. If that operand is a constant, there is nothing to reduce. Otherwise, it has to compute a value by applying the definitions of the functions (operators) used in that expression. This is how the evaluation of such an operand proceeds. The evaluator matches the pattern in each equation of the appropriate function until a matching pattern is found. Then it replaces the matched portion with the right side of that equation. Let us see how it evaluates `power2 3`.

```
   power2 3
=  2 * (power2 2)             -- apply function definition on 3
=  2 * (2 * (power2 1))       -- apply function definition on 2
=  2 * (2 * (2 * (power2 0))) -- apply function definition on 1
=  2 * (2 * (2 * (1)))        -- apply function definition on 0
=  2 * (2 * (2))              -- apply definition of *
=  2 * (4)                    -- apply definition of *
=  8                          -- apply definition of *
```

What is important to note is that each recursive call is made to a strictly *smaller* argument, and there is a *smallest* argument for which the function value is explicitly specified. In this case, numbers are compared by their magnitudes, and the smallest number is `0`. You will get an error in evaluating `power2 (-1)`. We will see more general recursive schemes in which there may be several *base* cases, and the call structure is more elaborate, but the simple scheme described here, called *primitive recursion*, is applicable in a large number of cases.

5.5.2 Counting the 1s in a Binary Expansion

Next, let us program a function whose value is the number of 1s in the binary expansion of its argument, where we assume that the argument is a natural number. Imagine scanning the binary expansion of a number from right to left (i.e., from the lower order to the higher order bits) starting at the lowest bit; if the current bit is 0, then we ignore it and move to the next higher bit, and if it is 1, then we add 1 to a running count (which is initially 0) and move to the next higher bit. Checking the lowest bit can be accomplished by the functions `even` and `odd`. Each successive bit can be accessed via integer division by 2 (right shift).

```
count 0 = 0
count n
 | even n = count (n `div` 2)
 | odd  n = count (n `div` 2) + 1
```

Note on pattern matching It would have been nice if we could have written the second equation as follows.

```
count 2*t     = count t
count 2*t + 1 = (count t) + 1
```

Unfortunately, Haskell does not allow such pattern matching.

5.5.3 Multiplication via Addition

Let us now implement multiplication using only addition. We make use of the identity $x * (y + 1) = x * y + x$.

```
mlt x 0     = 0
mlt x (y+1) = (mlt x y) + x
```

The recursive call is made to a strictly smaller argument in each case. There are two arguments which are both numbers, and the second number is strictly decreasing in each call. The smallest value of the arguments is attained when the second argument is 0.

The multiplication algorithm has a running time roughly proportional to the magnitude of `y`, because each call decreases `y` by 1. We now present a far better algorithm. You should study it carefully because it introduces an important concept, *generalizing the function*. The idea is that we write a function to calculate something more general, and then we call this function with a restricted set of arguments to calculate our desired answer. Let us write a function, `quickMlt`, that computes `x*y + z` over its three arguments. We can then define

```
mlt x y = quickMlt x y 0
```

The reason we define `quickMlt` is that it is more efficient to compute than `mlt` defined earlier. We will use the following result from arithmetic.

$$x \times (2 \times t) + z \;=\; (2 \times x) \times t + z$$
$$x \times (2 \times t + 1) + z \;=\; (2 \times x) \times t + (x + z)$$

The resulting program is:

```
quickMlt x 0 z = z
quickMlt x  y  z
 | even y = quickMlt (2 * x ) (y `div` 2) z
 | odd  y = quickMlt (2 * x ) (y `div` 2) (x + z)
```

In each case, again the second argument is strictly decreasing. In fact, it is being halved, so the running time is proportional to $\log y$.

Exercise 41

Use the strategy shown for multiplication to compute x^y. I suggest that you compute the more general function $z * x^y$. □

5.5.4 Fibonacci Numbers

The *Fibonacci sequence* (named after a famous Italian mathematician of the 10th century) is the sequence of integers whose first two terms are 0 and 1, and where each subsequent term is the sum of the previous two terms. So, the sequence starts out:

0 1 1 2 3 5 8 13 21 34 ...

Let us index the terms starting at 0, so the 0th fibonacci number is 0, the next one 1, and so forth. Our goal is to write a function that has argument n and returns the nth Fibonacci number. The style of programming applies to many others sequences in which each successive term is defined in terms of the previous ones. Note, particularly, the pattern matching applied in the last equation.

```
fib 0 = 0
fib 1 = 1
fib (n + 2) = (fib n) +  (fib (n+1))
```

Or, equivalently, we may write

```
fib n
 | n == 0 = 0
 | n == 1 = 1
 | otherwise = (fib (n-1)) +  (fib (n-2))
```

The first definition has three equations and the second has one conditional equation with three (guarded) clauses. Either definition works, but these programs are quite inefficient. Let us see how many times `fib` is called in computing `(fib 6)`, see Figure 5.1. Here each node is labeled with a number, the argument of a call on `fib`; the root node is labeled `6`. In computing `(fib 6)`, `(fib 4)` and `(fib 5)` have to be computed; so, the two children of `6` are `4` and `5`. In general, the children of node labeled `i+2` are `i` and `i+1`. As you can see, there is considerable recomputation; in fact, the computation time is proportional to the value being computed. (Note that `(fib 6)` `(fib 5)` `(fib 4)` `(fib 3)` `(fib 2)` `(fib 1)` are called 1 1 2 3 5 8 times, respectively, which is a part of the Fibonacci sequence). We will see a better strategy for computing Fibonacci numbers in the next section.

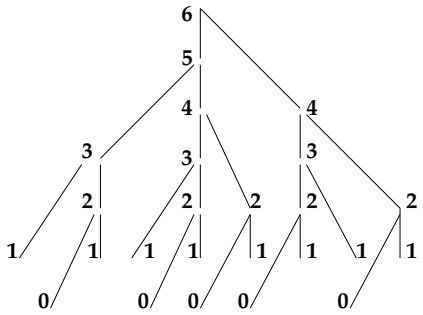


Figure 5.1: Call pattern in computing `(fib 6)`

5.5.5 Greatest Common Divisor

The greatest common divisor (gcd) of two positive integers is the largest positive integer that divides both m and n. (Prove the existence and uniqueness of gcd). Euclid gave an algorithm for computing gcd about 2,500 years ago, an algorithm that is still used today. Euclid's algorithm is as follows.

```
egcd m n
 | m == 0 = n
 | n == 0 = m
 | m == n = m
 | m > n  = egcd (m `rem` n) n
 | n > m  = egcd m (n `rem` m)
```

A simpler version of this algorithm, though not as efficient, is

```
gcd m n
 | m == n = m
 | m > n  = gcd (m - n) n
 | n > m  = gcd m (n - m)
```

This algorithm, essentially, computes the remainders, `(m `rem` n) and (n `rem` m)`, using repeated subtraction.

There is a modern version of the gcd algorithm, known as *binary gcd*. This algorithm uses multiplication and division by 2, which are implemented by shifts on binary numbers. The algorithm uses the following facts: (1) if `m == n`, then `gcd m n = m`, (2) if `m` and `n` are both even, say `2s` and `2t`, then `gcd m n = 2* (gcd s t)`, (3) if exactly one of `m` and `n`, say `m`, is even and equal to `2s`, then `gcd m n = gcd s n`, (5) if `m` and `n` are both odd (so `m-n` is even) and, say, `m > n`, then `gcd m n = gcd (m-n) n`.

```
bgcd m n
 | m == n               = m
 | (even m) && (even n) = 2 * (bgcd s t)
 | (even m) && (odd  n) =      bgcd s n
 | (odd  m) && (even n) =      bgcd m t
 | m > n                =      bgcd (m-n) n
 | n > m                =      bgcd m (n-m)
                        where s = m `div` 2
                              t = n `div` 2
```

Exercise 42
For positive integers m and n prove that

1. `gcd m n = gcd n m`
2. `gcd (m+n) n = gcd m n` □

5.6 Reasoning about Recursive Programs

We are interested in proving properties of programs, imperative and recursive, for similar reasons, to guarantee correctness, establish equivalence among alternative definitions, and, gain insight into the performance of the program. Our main tool in proving properties of functional programs is *induction*.

Note on Font Usage I will be using mathematical font instead of computer font —i.e., *power*2 instead of `power2`— in this section.

5.6.1 Proving Properties of *power*2

On page 102 we defined the function *power*2 that computes 2^n given n as input. We prove this result for all natural n. So, the desired proposition to be proven is, for all n, $n \geq 0$,

$P(n) :: power2\ n = 2^n$

We prove $P(n)$ by induction on n.

• $P(0) :: power2\ 0 = 2^0$: The left side, from the definition of *power*2, is 1; the right side, from arithmetic, is 1. Hence, $P(0)$.

• $P(n+1)$ given $P(n)$:

$$
\begin{array}{ll}
 & power2\ (n+1) \\
= & \{\text{definition of } power2\} \\
 & 2 * (power2\ n) \\
= & \{\text{induction hypothesis: } P(n) :: power2\ n = 2^n\} \\
 & 2 * (2^n) \\
= & \{\text{arithmetic}\} \\
 & 2^{n+1}
\end{array}
$$

Exercise 43

Prove the following facts about *power*2 directly from its definition (i.e., not using the fact that *power*2 $n = 2^n$). Below, n and m are arbitrary natural numbers.

1. $(power2\ n) > n$
2. $(power2\ (n+1)) > (power2\ n)$
3. $(power2\ (n+m)) = (power2\ n) * (power2\ m)$
4. $power2\ n = 3^n$. Where does the proof break down? □

5.6.2 Proving Properties of *count*

The example *power*2 of Section 5.6.1 was too easy; the property was obvious. Proofs are particularly important when the properties are not obvious. We will see one such proof for function *count* of page 102.

Unlike *power*2, *count* does not have a closed arithmetic form. In fact, this is true of most functions defined in a functional program. But we can still prove facts about such a function using induction.

We will prove that for non-negative integers m and n,

$$count(m+n) \leq (count\ m) + (count\ n)$$

The result is immediate if either m or n is zero. But the general result is not obvious. If you start proving this result by considering the addition process for binary numbers, you will be doing some of the steps of the proof given next. First, we note that we have two parameters, m and n, over which the property is defined. So, we will do induction over pairs of naturals and we will call a pair (p, q) *smaller* than (m, n) if $p < m$ and $q < n$.

Next, we will use the following properties of *count* without proof. You are asked to prove the first property in one of the exercises, the other two are mere rewritings of the definition. We use them because they are easier to manipulate in a proof than the expressions involving `‘div‘`.

$$count(0) = 0$$
$$count(2 \times x) = (count\ x)$$
$$count(2 \times x + 1) = (count\ x) + 1$$

We will consider four cases: $(even\ m) \wedge (even\ n)$, $(even\ m) \wedge (odd\ n)$, $(odd\ m) \wedge (even\ n)$, $(odd\ m) \wedge (odd\ n)$. The second and third cases are identical, because m and n are interchangeable due to symmetry in the problem statement. We will write $m = 2 \times s$ for $(even\ m)$ and $m = 2 \times s + 1$ for $(odd\ m)$, and similarly for n. In all cases, we apply induction on the magnitude of the numbers.

Here is the proof of $count(m+n) \leq (count\ m) + (count\ n)$ for $(even\ m) \wedge (even\ n)$: We have $m = 2 \times s$ and $n = 2 \times t$, for some s and t.

$$
\begin{array}{ll}
 & \mathit{count}(m+n) \\
= & \{m = 2 \times s \text{ and } n = 2 \times t\} \\
 & \mathit{count}(2 \times s + 2 \times t) \\
= & \{\text{arithmetic}\} \\
 & \mathit{count}(2 \times (s+t)) \\
= & \{\text{definition of } \mathit{count}\text{, given above}\} \\
 & \mathit{count}(s+t) \\
\leq & \{\text{induction hypothesis}\} \\
 & (\mathit{count}\ s) + (\mathit{count}\ t) \\
= & \{\text{definition of } \mathit{count}\text{, given above}\} \\
 & \mathit{count}(2 \times s) + \mathit{count}(2 \times t) \\
= & \{m = 2 \times s \text{ and } n = 2 \times t\} \\
 & \mathit{count}(m) + \mathit{count}(n)
\end{array}
$$

This is the easiest of the four proofs. Now, we prove $\mathit{count}(m+n) \leq (\mathit{count}\ m)+(\mathit{count}\ n)$ for $(\mathit{even}\ m)\wedge(\mathit{odd}\ n)$: We have $m = 2\times s$ and $n = 2\times t+1$, for some s and t.

$$
\begin{array}{ll}
 & \mathit{count}(m+n) \\
= & \{m = 2 \times s \text{ and } n = 2 \times t + 1\} \\
 & \mathit{count}(2 \times s + 2 \times t + 1) \\
= & \{\text{arithmetic}\} \\
 & \mathit{count}(2 \times (s+t) + 1) \\
= & \{\text{definition of } \mathit{count}\text{, given above}\} \\
 & \mathit{count}(s+t) + 1 \\
\leq & \{\text{induction hypothesis}\} \\
 & (\mathit{count}\ s) + (\mathit{count}\ t) + 1 \\
= & \{\text{definition of } \mathit{count}\text{, given above}\} \\
 & \mathit{count}(2 \times s) + \mathit{count}(2 \times t + 1) \\
= & \{m = 2 \times s \text{ and } n = 2 \times t + 1\} \\
 & \mathit{count}(m) + \mathit{count}(n)
\end{array}
$$

Now we prove $\mathit{count}(m+n) \leq (\mathit{count}\ m) + (\mathit{count}\ n)$ for $(\mathit{odd}\ m) \wedge (\mathit{odd}\ n)$: We have $m = 2 \times s + 1$ and $n = 2 \times t + 1$, for some s and t.

$$
\begin{array}{ll}
 & \mathit{count}(m+n) \\
= & \{m = 2 \times s + 1 \text{ and } n = 2 \times t + 1\} \\
 & \mathit{count}(2 \times s + 1 + 2 \times t + 1) \\
= & \{\text{arithmetic}\} \\
 & \mathit{count}(2 \times (s+t) + 2) \\
= & \{\text{definition of } \mathit{count}\text{, given above}\} \\
 & \mathit{count}(s + t + 1) \\
= & \{\text{regrouping the terms}\} \\
 & \mathit{count}((s) + (t+1)) \\
\leq & \{\text{induction hypothesis}\} \\
 & (\mathit{count}\ s) + (\mathit{count}\ (t+1)) \\
\leq & \{\text{induction hypothesis, applied to the second term}\}
\end{array}
$$

$$
\begin{array}{ll}
 & (count\ s) + (count\ t) + count(1) \\
< & \{\text{arithmetic; } count(1) = 1\} \\
 & ((count\ s) + 1) + ((count\ t) + 1) \\
= & \{\text{definition of } count\text{, given above}\} \\
 & count(2 \times s + 1) + count(2 \times t + 1) \\
= & \{m = 2 \times s + 1 \text{ and } n = 2 \times t + 1\} \\
 & count(m) + count(n)
\end{array}
$$

Exercise 44
Prove each of the following statements:

1. $count\ 2^n = 1$, for all n, $n \geq 0$,
2. $count\ (2^n - 1) = n$, for all n, $n > 0$,
3. $count\ (2^n + 1) = 2$, for all n, $n > 0$.
4. $count(power2\ n) = 1$, for all n, $n \geq 0$. Use only the definitions of *count* and *power*2; do not use the fact that $power2\ n = 2^n$. □

Exercise 45
This set of exercises are about *fib* and *gcd*, defined in Sections 5.5.4 (page 104) and 5.5.5 (page 105).

1. Prove that for all m and n, $m > 0$ and $n \geq 0$.

   $$fib(m + n) = fib(m - 1) \times fib(n) + fib(m) \times fib(n + 1).$$

2. Prove that

   $$gcd\ (fib\ n)\ (fib\ (n + 1)) = 1, \text{ for all } n,\ n > 0.$$

   There is a beautiful generalization of this result (it is slightly hard to prove); for all m and n, $m > 0$ and $n > 0$

   $$gcd\ (fib\ m)\ (fib\ n) = fib(gcd\ m\ n).$$

3. Show that during the computation of $(fib\ n)$, for any m, $0 < m \leq n$, $(fib\ m)$ is called $(fib\ (n + 1 - m))$ times. Use the fact that $(fib\ t)$ is called once for each call on $(fib\ (t + 1))$ and$(fib\ (t + 2))$. □

5.7 Tuple

We have, so far, seen a few elementary data types. There are two important ways we can build larger structures using data from the elementary types—*tuple* and *list*. We cover tuple in this section.

Tuple is the Haskell's version of a *record*; we may put several kinds of data together and give it a name. In the simplest case, we put together two pieces of data and form a 2-tuple, also called a *pair*. Here are some examples.

```
(3,5)   ("Misra","Jayadev")  (True,3)  (False,0)
```

As you can see, the components may be of different types. Note that Haskell treats `(3)` and `3` alike, both are treated as numbers. Let us add the following definitions to our program file:

```
teacher = ("Misra","Jayadev")
uniqno = 59285
course = ("cs337",uniqno)
```

There are two predefined functions, *fst* and *snd*, that return the first and the second components of a pair, respectively.

```
Main> fst(3,5)
3
Main> snd teacher
"Jayadev"
Main> snd course
59285
```

There is no restriction at all in Haskell about what you can have as the first and the second component of a tuple. In particular, we can create another tuple

```
hard = (teacher,course)
```

and extract its first and second components by

```
Main> fst hard
("Misra","Jayadev")
Main> snd hard
("cs337",59285)
```

Haskell allows you to create tuples with any number of components, but *fst* and *snd* are applicable only to a pair.

5.7.1 Revisiting the Fibonacci Computation

As we saw in the Figure 5.1 of page 105, there is considerable recomputation in evaluating `fib n`, in general. Here, I sketch a strategy to eliminate the recomputation. We define a function, called `fibpair`, which has an argument `n`, and returns the pair `((fib n), (fib (n+1)))`, i.e., two consecutive elements of the Fibonacci sequence. This function can be computed efficiently, as shown below, and we may define `fib n = fst(fibpair n)`.

```
fibpair 0 = (0,1)
fibpair n = (y, x+y)
             where (x,y) = fibpair (n-1)
```

5.7.2 A Fundamental Theorem of Number Theory

The following theorem, due to Euclid, is fundamental in number theory. Given positive integers m and n, there exist integers a and b such that

$$a \times m + b \times n = \gcd(m, n).$$

We prove this result by induction on pairs of positive integers. Order such pairs lexicographically. We prove a more general base case than the case for the pair $(1, 1)$.

• $m = n$:

$$\begin{aligned} & 1 \times m + 0 \times n \\ = \; & m \\ = \; & \gcd(m, m) \end{aligned}$$

Thus, the theorem is established by letting $a, b = 1, 0$.

• Inductive Case: Assume that the result holds for all pairs that are lexicographically smaller than (m, n). We prove the result for (m, n). If $m = n$ the result holds from the base case, above. Therefore, we need only consider $m \neq n$. Assume that $m > n$; the analysis is symmetric for $n > m$.

Consider the pair of positive integers $(m - n, n)$ which is lexicographically smaller than (m, n). By the induction hypothesis, there exist integers a', b' such that

$$\begin{aligned} & a' \times (m - n) + b' \times n = \gcd((m - n), n) \\ \Rightarrow \quad & \{m > n \Rightarrow [\gcd((m - n), n) = \gcd(m, n)]\} \\ & a' \times (m - n) + b' \times n = \gcd(m, n) \\ \Rightarrow \quad & \{\text{rewriting the LHS}\} \\ & a' \times m + (b' - a') \times n = \gcd(m, n) \end{aligned}$$

Letting $a, b = a', b' - a'$, we have established the result for (m, n). You will be asked to write a program to compute a and b.

Exercise 46

1. What is the difference between `(3,4,5)` and `(3,(4,5))`?

2. A point in two dimensions is a pair of coordinates; assume that we are dealing with only integer coordinates. Write a function that takes two points as arguments and returns `True` iff either the $x-$ or the $y-$coordinate of the points are equal. Here is what I expect (`ray` is the name of the function).

```
Main> ray (3,5) (3,8)
True
Main> ray (3,5) (2,5)
```

```
True
Main> ray (3,5) (3,5)
True
Main> ray (3,5) (2,8)
False
```

3. A line is given by a pair of distinct points (its end points). Define function `parallel` that has two lines as arguments and value `True` iff they are parallel. Recall from coordinate geometry that two lines are parallel if their slopes are equal, and the slope of a line is given by the difference of the $y-$coordinates of its two points divided by the difference of their $x-$coordinates. In case the $x-$coordinates of two points of a line are equal, use a different strategy to decide if it is parallel to the other line.

   Note: You need the division operator, written as `/`, which yields a real value. Watch out for division by zero.

   Here is the result of some evaluations.

```
Main> parallel ((3,5), (3,8)) ((3,5), (3,7))
True
Main> parallel ((3,5), (4,8)) ((4,5), (3,7))
False
Main> parallel ((3,5), (4,7)) ((2,9), (0,5))
True
```

   Solution

```
parallel ((a,b),(c,d)) ((u,v),(x,y))
  | (a == c) || (u  == x) = (a == c) && (u  == x)
  | otherwise             =
         (d - b)/(c - a) == (y - v)/(x - u)
```

   Note that we do not expect both `(d - b)` and `(c - a)` to be zero, because the points `(a,b)` and `(c,d)` are distinct.

4. The function `fibpair n` returns the pair `((fib n), (fib (n+1)))`. The computation of `(fib (n+1))` is unnecessary, since we are interested only in `fib n`. Redefine `fib` so that this additional computation is avoided.

   Solution

```
fib 0 = 0
fib n = snd(fibpair (n-1))
```

5. Prove the result in Section 5.7.2 by ordering pairs of positive integers by the magnitude of their sums.

6. Use the arguments of Section 5.7.2 to compute a and b, given positive integers m and n, such that

$$a \times m + b \times n = \gcd(m, n)$$ □

5.8 Type

Every expression in Haskell has a *type*. The type may be specified by the programmer, or deduced by the interpreter. If you write `3+4`, the interpreter can deduce the type of the operands and the computed value to be integer (not quite, as you will see). When you define

```
imply p q = not p || q
digit c = ('0' <= c) && (c <= '9')
```

the interpreter can figure out that p and q in the first line are booleans (because || is applied only to booleans) and the result is also a boolean. In the second line, it deduces that c is a character because of the two comparisons in the right side, and that the value is boolean, from the types of the operands.

The type of an expression may be a primitive one: `Int`, `Bool`, `Char` or `String`, or a structured type, as explained below. You can ask to see the type of an expression by giving the command `:t`, as in the following.

```
Main> :t ('0' <= '9')
'0' <= '9' :: Bool
```

The type of a tuple is a tuple of types, one entry for the type of each operand. In the following, `[Char]` denotes a string; I will explain why in the next section.

```
Main> :t ("Misra","Jayadev")
("Misra","Jayadev") :: ([Char],[Char])
Main> :t teacher
teacher :: ([Char],[Char])
Main> :t course
course :: ([Char],Integer)
Main> :t (teacher,course)
(teacher,course) :: (([Char],[Char]),([Char],Integer))
```

Each function has a type, namely, the types of its arguments in order followed by the type of the result, all separated by `->`.

```
Main> :t imply
imply :: Bool -> Bool -> Bool
Main> :t digit
digit :: Char -> Bool
```

Capitalizations for types Type names (e.g., `Int`, `Bool`) are always capitalized. The name of a function or parameter should never be capitalized.

5.8.1 Polymorphism

Haskell allows us to write functions whose arguments can be *any* type, or any type that satisfies some constraint. Consider the *identity* function:

```
identity x = x
```

This function's type is

```
Main> :t identity
identity :: a -> a
```

That is for any type `a`, it accepts an argument of type `a` and returns a value of type `a`.

A less trivial example is a function whose argument is a pair and whose value is the same pair with its components exchanged.

```
exch (x,y) = (y,x)
```

Its type is as follows:

```
Main> :t exch
exch :: (a,b) -> (b,a)
```

Here, `a` and `b` are arbitrary types. So, `exch(3,5), exch (3,"misra"), exch ((2,'a'),5)` and `exch(exch ((2,'a'),5))` are all valid expressions. The interpreter chooses the most general type for a function so that the widest range of arguments would be accepted.

Now, consider a function whose argument is a pair and whose value is `True` iff the components of the pair are equal.

```
eqpair (x,y) = x == y
```

It is obvious that `eqpair (3,5)` makes sense, but not `eqpair (3,'j')`. We would expect the type of `eqpair` to be `(a,a) -> Bool`, but it is more subtle.

```
Main> :t eqpair
eqpair :: Eq a => (a,a) -> Bool
```

This says that the type of `eqpair` is `(a,a) -> Bool`, *for any type* `a` *that belongs to the* `Eq` *class, i.e., types over which* `==` *is defined*. Otherwise, the test `==` in `eqpair` cannot be performed. Equality is not necessarily defined on all types, particularly on function types.

Finally, consider a function that sorts two numbers which are given as a pair.

```
sort (x,y)
 | x <= y = (x,y)
 | x >  y = (y,x)
```

The type of `sort` is

```
Main> :t sort
sort :: Ord a => (a,a) -> (a,a)
```

It says that sort accepts any pair of elements of the same type, provided the type belongs to the `Ord` type class, i.e., there is an order relation defined over that type; `sort` returns a pair of the same type as its arguments. An order relation is defined over most of the primitive types. So we can do the following kinds of sorting. Note, particularly, the last example.

```
Main> sort (5,2)
(2,5)
Main> sort ('g','j')
('g','j')
Main> sort ("Misra", "Jayadev")
("Jayadev","Misra")
Main> sort (True, False)
(False,True)
Main> sort ((5,3),(3,4))
((3,4),(5,3))
```

Polymorphism means that a function can accept and produce data of many different types. This allows us to define a single sorting function, for example, which can be applied in a very general fashion. We will see later that we can sort a pair of *trees* using this sorting function.

5.8.2 Type Classes

Haskell has an extensive type system, which we will not cover in this course. Beyond types are type classes, which provide a convenient treatment of overloading. A type class is a collection of types, each of which has a certain function (or set of functions) defined on it. Here are several examples of type classes: the `Eq` class consists of all types on which an equality operation is defined; the `Ord` class consists of all types on which an order relation is defined; the `Num` class consists of all types on which typical arithmetic operations (`+, *,` etc.) are defined. The following exchange is instructive.

```
Main> :t 3
3 :: Num a => a
Main> :t (3,5)
(3,5) :: (Num a, Num b) => (b,a)
```

Read the second line to mean 3 has the type `a`, where `a` is any type in the type class `Num`. The last line says that `(3,5)` has the type `(b,a)`, where `a` and `b` are arbitrary (and possibly equal) types in the type class `Num`. So, what is the type of `3+4`? It has the type of any member of the `Num` class.

```
Main> :t 3+4
3 + 4 :: Num a => a
```

5.8.3 Type Violation

Since the interpreter can deduce the type of each expression, it can figure out if you have supplied the arguments of the right type for a function. If you provide invalid arguments, you will see something like this.

```
Main> digit 9
ERROR - Illegal Haskell 98 class constraint in inferred type
*** Expression : digit 9
*** Type       : Num Char => Bool

Main> imply True 3
ERROR - Illegal Haskell 98 class constraint in inferred type
*** Expression : imply True 3
*** Type       : Num Bool => Bool
```

5.9 List

Each tuple has a bounded number of components —two each for `course` and `teacher` and two in `(teacher,course)`. In order to process larger amounts of data, where the number of data items may not be known a priori, we use the data structure *list*. A list consists of a finite sequence of items[2] **all of the same type**. Here are some lists.

```
[1,3,5,7,9]   -- all odd numbers below 10
[2,3,5,7]     -- all primes below 10
[[2],[3],[5],[7]]   -- a list of lists
[(3,5), (3,8), (3,5), (3,7)] -- a list of tuples
[[(3,5), (3,8)], [(3,5), (3,7), (2,9)]] -- a list of list of tuples
['a','b','c'] -- a list of characters
["misra", "Jayadev"] ---- a list of strings
```

The following are not lists because not all their elements are of the same type.

[2]We deal with only finite lists in this note. Haskell permits definitions of infinite lists and computations on them, though only a finite portion can be computed in any invocation of a function.

```
[[2],3,5,7]
[(3,5), 8]
[(3,5), (3,8,2)]
['J',"misra"]
```

The order and number of elements in a list matter. So,

`[2,3]` $\neq$ `[3,2]`
`[2]` $\neq$ `[2,2]`

5.9.1 The Type of a List

The type of any list is `[ItemType]` where `ItemType` is the type of one of its items. So, `[True]` is a list of booleans and so are `[True, False]` and `[True, False, True, False]`. Any function that accepts a list of booleans as arguments can process any of these three lists. Here are these and some more examples.

```
Main> :t [True]
[True] :: [Bool]
Main> :t [True, False]
[True,False] :: [Bool]
Main> :t [(2,'c'), (3,'d')]
[(2,'c'),(3,'d')] :: Num a => [(a,Char)]
Main> :t [[2],[3],[5],[7]]
[[2],[3],[5],[7]] :: Num a => [[a]]
Main> :t [(3,5), (3,8), (3,5), (3,7)]
[(3,5),(3,8),(3,5),(3,7)] :: (Num a, Num b) => [(a,b)]
Main> :t [[(3,5), (3,8)], [(3,5), (3,7), (2,9)]]
[[(3,5),(3,8)],[(3,5),(3,7),(2,9)]] :: (Num a, Num b) => [[(b,a)]]
Main> :t ['a','b','c']
['a','b','c'] :: [Char]
```

A string is a list of characters, i.e., `[Char]`; each of its characters is taken to be a list item. Therefore, a list whose items are strings is a list of `[Char]`, or `[[Char]]`.

```
Main> :t ["misra"]
["misra"] :: [[Char]]
```

Empty List A very special case is an empty list, one having no items. We write it as `[]`. It appears a great deal in programming. What is the type of `[]`?

```
Main> :t []
[] :: [a]
```

This says that `[]` is a list of `a`, where `a` is *any* type. Therefore, `[]` can be given as argument wherever a list is expected.

5.9.2 The List Constructor *Cons*

There is one built-in operator that is used to construct a list element by element; it is pronounced *Cons* and is written as `:` (a colon). Consider the expression `x:xs`, where `x` is an item and `xs` is a list. The value of this expression is a list obtained by prepending `x` to `xs`. Note that `x` should have the same type as the items in `xs`. Here are some examples.

```
Main> 3: [2,1]
[3,2,1]
Main> 3: []
[3]
Main> 1: (2: (3: [])) --Study this one carefully.
[1,2,3]
Main> 'j': "misra"
"jmisra"
Main> "j": "misra"
ERROR - Type error in application
*** Expression     : "j" : "misra"
*** Term           : "j"
*** Type           : String
*** Does not match : Char
```

5.9.3 Pattern Matching on Lists

When dealing with lists, we often need to handle the special case of the empty list in a different manner. Pattern matching can be applied very effectively in such situations.

Let us consider a function `len` on lists that returns the length of the argument list. We need to differentiate between two cases, as shown below.

```
len [] = ..
len (x:xs) = ..
```

The definition of this function spans more than one equation. During function evaluation with a specific argument —say, `[1,2,3]`— each of the equations is checked from top to bottom to find the first one where the given list matches the pattern of the argument. So, with `[1,2,3]`, the first equation does not match because the argument is not an empty list. The second equation matches because `x` matches with `1` and `xs` matches with `[2,3]`. Additionally, pattern matching assigns names to components of the data structure —`x` and `xs` in this example— which may then be used in the RHS of the function definition.

5.9.4 Recursive Programming on Lists

Let us try to complete the definition of the function `len` sketched above. Clearly, we expect an empty list to have length 0. The general case, below, should be studied very carefully.

```
len [] = 0
len (x:xs) = 1 + (len xs)
```

The RHS of the second equation says that the length of the list `(x:xs)` is one more than the length of `xs`. This is surely true, but how does a computer use this to evaluate the result for a specific list like `[1,2,3]`? Here is a very rough explanation.

The interpreter has an expression to evaluate at any time. If all the operands in that expression are constants, it can easily evaluate it. If some operands are not constants, they are function calls, which are then expanded. So, this is how `len [1,2,3]` is evaluated. Recall that `[1,2,3]` is `(1:(2:(3:[])))`.

```
   len [1,2,3]
=  len (1:(2:(3:[])))   -- [1,2,3] is (1:(2:(3:[])))
=  1 + len (2:(3:[]))   -- applying function definition on [1,2,3]
=  1 + (1 + len (3:[])) -- applying function definition on [2,3]
=  1 + (1 + (1+ len []) -- applying function definition on [3]
=  1 + (1 + (1+ 0)      -- applying function definition on []
=  3                    -- reducing the expression
```

What is important to note is that each recursive call should be made to a *smaller* argument, and there should be a *smallest* argument for which the function value is explicitly specified. In our case, a list is *smaller* than another if the former has fewer elements, and since `xs` is smaller than `x:xs` our required criterion is met. The smallest list, by this measure, is the empty list, for which we have an explicit function value 0. This is, in fact, the strategy you will use in most of your programs.

Consider now a function that sums the elements of a list of integers. It follows the same pattern.

```
suml [] = 0
suml (x:xs) = x + (suml xs)
```

A function that multiplies the elements of a list of integers.

```
multl [] = 1
multl (x:xs) = x * (multl xs)
```

Next, we write a program for a function whose value is the maximum of a list of integers. Here it does not make much sense to talk about the maximum over an empty list.[3] So, our smallest list will have a single element, and here is how you pattern match for a single element.

```
maxl [x] = x
maxl (x:xs) = max x (maxl xs)
```

[3]But people do and they define it to be $-\infty$. The value $-\infty$ is approximated by the smallest value in type `Int` which is `minBound::Int`; similarly, $+\infty$ is approximated by the largest value, `maxBound::Int`.

Exercise 47
Write a function that takes the conjunction (`&&`) of the elements of a list of booleans.

```
andl [] = True
andl (x:xs) = x && (andl xs)
```

So, we have

```
Main> andl [True, True, 2 == 5]
False
```

□

Now consider a function whose value is not just one item but a list. The following function negates every entry of a list of booleans.

```
notl[] = []
notl (x:xs) = (not x) : (notl xs)
```

So,

```
Main> notl [True, True, 2 == 5]
[False,False,True]
```

The following function removes all negative numbers from the argument list.

```
negrem [] = []
negrem (x:xs)
 | x < 0     = negrem xs
 | otherwise = x : (negrem xs)
```

So,

```
Main> negrem []
[]
Main> negrem [2,-3,1]
[2,1]
Main> negrem [-2,-3,-1]
[]
```

Pattern matching over a list may be quite involved. The following function, `divd`, partitions the elements of the argument list between two lists, putting the elements with even index in the first list and with odd index in the second list (list elements are numbered starting at 0). So, `divd [1,2,3]` is `([1,3],[2])` and `divd [1,2,3,4]` is `([1,3],[2,4])`. See Section 5.9.5 for another solution to this problem.

```
divd [] = ([], [])
divd (x: xs) = (x:ys, zs)
                where (zs,ys) = divd xs
```

We conclude this section with a small example that goes beyond "primitive recursion", i.e., recursion is applied not just on the tail of the list. The problem is to define a function `uniq` that returns the list of unique items from the argument list. So, `uniq[3, 2, 2, 2] = [3, 2]`, `uniq[3, 2, 2] = [3, 2]` and `uniq[3, 2] = [3, 2]`.

```
uniq [] = []
uniq (x:xs) = x: (uniq(minus x xs))
              where
                    minus y [] = []
                    minus y (z: ys)
                     | y == z    = (minus y ys)
                     | otherwise = z: (minus y ys)
```

Note This program does not work if you try to evaluate `uniq []` on the command line. This has to do with type classes; the full explanation is beyond the scope of these notes.

Exercise 48

1. Define a function `unq` that takes two lists `xs` and `ys` as arguments. Assume that initially `ys` contains distinct elements. Function `unq` returns the list of unique elements from `xs` and `ys`. Define `uniq` using `unq`.

   ```
   unq [] ys     = ys
   unq (x:xs) ys
    | inq x ys   = unq xs ys -- inq x ys is: x in ys?
    | otherwise  = unq xs (x:ys)
                     where
                      inq y []      = False
                      inq y (z: zs) = (y == z) || (inq y zs)
   uniq xs = unq xs []
   ```

2. Define a function that creates a list of unique elements from a sorted list. So, a possible input is `[2,2,3,3,4]` and the corresponding output is `[2,3,4]`.

3. The *prefix sum* of a list of numbers is a list of equal length whose ith element is the sum of the first i items of the original list. So, the prefix sum of [3,1,7] is [3,4,11]. Write a linear-time algorithm to compute the prefix sum.
   Hint: Use function generalization.

   ```
   ps xs = pt xs 0
           where pt [] c = []
                 pt (x:xs) c = (c+x) : (pt xs (c+x))
   ```

5.9.5 Mutual Recursion

All of our examples so far have involved recursion in which a function calls itself. It is easy to extend this concept to a group of functions that call each other. To illustrate mutual recursion, I will consider the problem of partitioning a list; see page 120 for another solution to this problem.

It is required to create two lists of nearly equal size from a given list, *lis*. The order of items in *lis* is irrelevant, so the two created lists may contain elements from *lis* in arbitrary order. If *lis* has an even number of elements, say $2 \times n$, then each of the created lists has n elements, and if *lis* has $2 \times n + 1$ items, one of the lists has $n + 1$ elements and the other has n elements.

One possible solution for this problem is to determine the length of *lis* (you may use the built-in function `length`) and then march down *lis* half way, adding elements to one output list, and then continue to the end of *lis* adding items to the second output list. We adopt a simpler strategy. We march down *lis*, adding items alternately into the two output lists. We define two functions, `divide0` and `divide1` each of which partitions the argument list, `divide0` starts with prepending the first item of the argument into the first list, and `divide1` by prepending the first item to the second list. Here, `divide0` calls `divide1` and `divide1` calls `divide0`.

```
divide0 []      = ([],[])
divide0 (x: xs) = (x:f, s)
                   where (f,s) = divide1 xs

divide1 []      = ([],[])
divide1 (x: xs) = (f, x:s)
                   where (f,s) = divide0 xs
```

We then get,

```
Main> divide0 [1,2,3]
([1,3],[2])
Main> divide0 [1,2,3,4]
([1,3],[2,4])
```

5.10 Examples of Programming with Lists

In this section, we take up more elaborate examples of list-based programming.

5.10.1 Some Useful List Operations

`snoc`

The list constructor `cons` of Section 5.9.2 (page 118) is used to add an item at the head of a list. The function `snoc`, defined below, adds an item at the "end" of a list.

```
snoc x []       = [x]
snoc x (y: xs) = y:(snoc x xs)
```

The execution of `snoc` takes time proportional to the length of the argument list, whereas `cons` takes constant time. So, it is preferable to use `cons`.

`concatenate`

The following function concatenates two lists in order. Remember that the two lists need to have the same type in order to be concatenated.

```
conc [] ys = ys
conc (x:xs) ys = x : (conc xs ys)
```

There is a built-in operator that does the same job; `conc xs ys` is written as `xs ++ ys`. The execution of `conc` takes time proportional to the length of the first argument list.

Exercise 49

1. Implement a double-ended queue in which items may be added at either end and removed from either end.
2. Define a function to left-rotate a list. Left-rotation of `[1,2,3]` yields `[2,3,1]` and of the empty list yields the empty list. □

`flatten`

Function `flatten` takes a list of lists, like

```
[ [1,2,3], [10,20], [], [30] ]
```

and flattens it out by putting all the elements into a single list, like

```
[1,2,3,10,20,30]
```

This definition should be studied carefully. Here `xs` is a list and `xss` is a list of lists.

```
flatten [] = []
flatten (xs : xss) = xs ++ (flatten xss)
```

Exercise 50

1. What is the type of `flatten`?
2. Evaluate

   ```
   ["I"," love"," functional"," programming"]
   ```

   and

   ```
   flatten ["I"," love"," functional"," programming"]
   ```

   and note the difference.
3. What happens if you apply `flatten` to a list of list of lists? □

`reverse`

The following function reverses the order of the items in a list.

```
rev []      = []
rev (x: xs) = (rev xs) ++ [x]  -- put x at the end of (rev xs)
```

The running time of this algorithm is $O(n^2)$, where n is the length of the argument list. (Prove this result using a recurrence relation. Use the fact that `append` takes $O(k)$ time when applied to a list of length k.) In the imperative style, reversing of an array can be done in linear time. Something is terribly wrong with functional programming! Actually, we *can* attain a linear time bound using functional programming.

The more efficient algorithm uses *function generalization* which was introduced for the `quickMlt` function for multiplication example in Section 5.5.3 (page 103). We define a function `reverse` that has two arguments `xs` and `ys`, each a list. Here `xs` denotes the part that remains to be reversed (a suffix of the original list) and `ys` is the reversal of the prefix. So, during the computation of `reverse [1,2,3,4,5]`, there will be a call to `reverse [4,5] [3,2,1]`. We have the identity

```
reverse xs ys = (rev xs) ++ ys
```

Given this identity,

```
reverse xs [] = rev xs
```

The definition of `reverse` is as follows.

```
reverse [] ys     = ys
reverse (x:xs) ys = reverse xs (x:ys)
```

Exercise 51

1. Show that the execution time of `reverse xs ys` is $O(n)$ where the length of `xs` is n.
2. Prove from the definition of `rev` that `rev (rev xs) = xs`.
3. Prove from the definition of `rev` and `reverse` that

   ```
   reverse xs ys = (rev xs) ++ ys
   ```

4. Show how to right-rotate a list efficiently (i.e., in linear time in the size of the argument list). Right-rotation of `[1,2,3,4]` yields `[4,1,2,3]`.
   Hint: use `rev`.

   ```
   right_rotate []    = []
   right_rotate xs    = y: (rev ys)
                        where
                        y:ys = (rev xs)
   ```

5. Try proving `rev (xs ++ ys) = (rev ys) ++ (rev xs)`. □

5.10.2 Towers of Hanoi

This is a well-known puzzle. Given is a board on which there are three pegs marked `'a', 'b' and 'c'`, on each of which can rest a stack of disks. There are n disks, $n > 0$, of varying sizes, numbered 1 through n in order of size. The disks are *correctly stacked* if they are increasing in size from top to bottom in each stack. Initially, all disks are correctly stacked at `'a'`. It is required to move all the disks to `'c'` in a sequence of steps under the constraints that (1) in each step, the top disk of one stack is moved to the top of another stack, and (2) the disks are correctly stacked at all times.

For $n = 3$, the sequence of steps given below is sufficient. In this sequence, a triple `(i,x,y)` denotes a step in which disk `i` is moved from stack `x` to `y`. Clearly, `i` is at the top of stack `x` before the step and at the top of stack `y` after the step.

```
[(1,'a','b'),(2,'a','c'),(1,'b','c'),(3,'a','b'),
 (1,'c','a'),(2,'c','b'),(1,'a','b')]
```

There is an iterative solution for this problem, which goes like this. Disk 1 moves in every alternate step starting with the first step. If n is odd, disk 1 moves cyclically from `'a'` to `'b'` to `'c'` to `'a'` ..., and if n is even, disk 1 moves cyclically from `'a'` to `'c'` to `'b'` to `'a'` .... In each remaining step, there is exactly one possible move: ignore the stack of which disk 1 is the top; compare the tops of the two remaining stacks and move the smaller one to the top of the other stack (if one stack is empty, move the top of the other stack to its top).

I don't know an easy proof of this iterative scheme; in fact, the best proof I know shows that this scheme is equivalent to an obviously correct recursive scheme.

The recursive scheme is based on the following observations. There is a step in which the largest disk is moved from `'a'`; we show that it is sufficient to move it only once, from `'a'` to `'b'`. At that moment, disk n is the top disk at `'a'` and there is no other disk at `'b'`. So, all other disks are at `'c'`, and, according to the given constraint, they are correctly stacked. Therefore, prior to the move of disk n, we have the subtask of moving the remaining $n-1$ disks, provided $n > 1$, from `'a'` to `'c'`. Following the move of disk n, the subtask is to move the remaining $n-1$ disks from `'a'` to `'c'`. Each of these subtasks is smaller than the original task, and may be solved recursively. Note that in solving the subtasks, disk n may be disregarded, because any disk can be placed on it; hence, its presence or absence is immaterial.

```
tower n a b c
 | n == 0    = []
 | otherwise =    (tower (n-1) a c b)
               ++ [(n,a,b)]
               ++ (tower (n-1) c b a)
```

We get

```
Main> tower 3 'a' 'b' 'c'
[(1,'a','b'),(2,'a','c'),(1,'b','c'),(3,'a','b'),
 (1,'c','a'),(2,'c','b'),(1,'a','b')]
```

Exercise 52

1. What is the type of function `tower`?
2. What is the total number of moves, as a function of n?
3. Argue that there is no scheme that uses fewer moves, for any n.
4. Show that disk 1 is moved in every alternate move.
5. (very hard) What is a good strategy (i.e., minimizing the number of moves) when there are four pegs instead of three?
6. (Gray code; hard) Start with an n-bit string of all zeros. Number the bits 1 through n, from lower to higher bits. Solve the Towers of Hanoi problem for n, and whenever disk i is moved, flip the ith bit of your number and record it. Show that all 2^n n-bit strings are recorded exactly once in this procedure. □

5.10.3 Gray Code

If you are asked to list all 3-bit numbers, you will probably write them in increasing order of their magnitudes:

000 001 010 011 100 101 110 111

There is another way to list these numbers so that consecutive numbers (the first and the last numbers are consecutive too) differ in exactly one bit position.

000 001 011 010 110 111 101 100

The problem is to generate such a sequence for every n. Let us attack the problem by induction on n. For $n = 1$, the sequence 0 1 certainly meets the criterion. For $n+1$, $n \geq 1$, we argue as follows. Assume, inductively, that there is a sequence X_n of n-bit numbers in which the consecutive numbers differ in exactly one bit position. Now, we will take X_n and create X_{n+1}, a sequence of $n+1$-bit numbers with the same property. To gain some intuition, let us look at X_2. Here is a possible sequence:

00 01 11 10

How do we construct X_3 from X_2? Appending the same bit, a 0 or a 1, to the left end of each bit string in X_2 preserves the property among consecutive numbers, and it makes each bit string longer by one. So, from the above sequence we get:

000 001 011 010

Now, we need to list the remaining 3-bit numbers, which we get by appending 1s to the sequence X_2:

100 101 111 110

But merely concatenating this sequence to the previous sequence won't do; 010 and 100 differ in more than one bit position. But concatenating the reverse of the above sequence works, and we get

000 001 011 010 110 111 101 100

Define function `gray` to compute such a sequence given n as the argument. The output of the function is a list of 2^n items, where each item is a n-bit string. Thus, the output will be

```
Main> gray 3
["000","001","011","010","110","111","101","100"]
```

Considering that we will have to reverse this list to compute the function value for the next higher argument, let us define a more general function, `grayGen`, whose argument is a natural number n and whose output is a pair of lists, `(xs,ys)`, where `xs` is the Gray code of n and `ys` is the reverse of `xs`. We can compute `xs` and `ys` in similar ways, without actually applying the reverse operation.

First, define a function `cons0` whose argument is a list of strings and which returns a list by prepending a '0' to each string in the argument list. Similarly, define `cons1` which prepends '1' to each string.

```
cons0 [] = []
cons0 (x:xs) = ('0':x):(cons0 xs)

cons1 [] = []
cons1 (x:xs) = ('1':x):(cons1 xs)
```

Then `grayGen` and `gray` are easy to define.

```
grayGen 0     = ([""],[""])
grayGen (n+1) = ((cons0 a) ++ (cons1 b), (cons1 a) ++ (cons0 b))
                  where (a,b) = grayGen n

gray n = fst(grayGen n)
```

Exercise 53

1. Show another sequence of 3-bit numbers that has the Gray code property.
2. Prove that for all n,

```
rev a = b
      where (a,b) = grayGen n
```

You will have to use the following facts; for arbitrary lists `xs` and `ys`

```
rev (rev xs)   = xs
rev (cons0 ys) = cons0 (rev ys)
rev (cons1 ys) = cons1 (rev ys)
```

3. Given two strings of equal length, their *Hamming distance* is the number of positions in which they differ. Define a function to compute the Hamming distance of two given strings.

4. In a Gray code sequence consecutive numbers have hamming distance of 1. Write a function that determines if the strings in its argument list have the Gray code property. Make sure that you compare the first and last elements of the list. □

5.10.4 Sorting

Consider a list of items drawn from some totally ordered domain such as the integers. We develop a number of algorithms for sorting such a list, that is, for producing a list in which the same set of numbers are arranged in ascending order.[4] We cannot do in situ exchanges in sorting, as is typically done in imperative programming, because there is no way to modify the argument list.

Insertion Sort

Using the familiar strategy of primitive recursion, let us define a function for sorting, as follows.

```
isort []     = []
isort (x:xs) =  .. (isort xs) .. -- skeleton of a definition
```

The first line is easy to justify. For the second line, the question is: how can we get the sorted version of `(x:xs)` from the sorted version of `xs` —that is `isort xs`— and `x`? The answer is, insert `x` at the right place in `(isort xs)`. So, let us first define a function `insert y ys`, which produces a sorted list by appropriately inserting `y` in the sorted list `ys`.

```
insert y [] = [y]
insert y (z:zs)
 | y <= z   = y:(z:zs)
 | y > z    = z: (insert y zs)
```

[4] A sequence of numbers $\ldots x\ y \ldots$ is *ascending* if for consecutive elements x and y, $x \leq y$ and *increasing* if $x < y$. It is *descending* if $x \geq y$ and *decreasing* if $x > y$.

Then, function `isort` is

```
isort []     = []
isort (x:xs) = insert x (isort xs)
```

Exercise 54

1. What is the worst-case running time of `insert` and `isort`?
2. What is the worst-case running time of `isort` if the input list is already sorted? What if the reverse of the input list is sorted (i.e., the input list is sorted in descending order)? □

Merge sort

This sorting strategy is based on merging two lists. First, we divide the input list into two lists of nearly equal size —function `divide0` of Section 5.9.5 (page 122) works very well for this— sort the two lists recursively and then merge them. Merging of sorted lists is easy; see function `merge` below.

```
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys)
 | x <= y   = x : (merge xs (y:ys))
 | x >  y   = y : (merge (x:xs) ys)
```

Based on this function, we develop `mergesort`.

```
mergesort [] = []
mergesort [x] = [x]
mergesort xs = merge left right
               where
                (xsl,xsr) = divide0 xs
                left      = mergesort xsl
                right     = mergesort xsr
```

Exercise 55

1. Why is

   ```
   merge [] [] = []
   ```

   not a part of the definition of `merge`?
2. Show that mergesort has a running time of $O(2^n \times n)$ where the argument list has length 2^n.
3. Modify `merge` so that it discards all duplicate items.

4. Develop a function similar to `merge` that has two ascending lists as arguments and creates an ascending list of common elements.

5. Develop a function that has two ascending lists as arguments and creates the difference, first list minus the second list, as an ascending list of elements.

6. Develop a function that has two ascending lists of integers as arguments and creates an increasing list of pairwise sums from the two lists (duplicates are discarded). □

Quicksort

Function `quicksort` partitions its input list `xs` into two lists, `ys` and `zs`, so that every item of `ys` is at most every item of `zs`. Then `ys` and `zs` are sorted and concatenated. Note that in `mergesort`, the initial partitioning is easy and the final combination is where the work takes place; in `quicksort` the initial partitioning is where all the work is.

We develop a version of `quicksort` that differs slightly from the description given above. First, we consider the partitioning problem. A list is partitioned with respect to some value `v` that is supplied as an argument; all items smaller than or equal to `v` are put in `ys` and all items greater than `v` are put in `zs`.

```
partition v []   = ([],[])
partition v (x:xs)
 | x <= v = ((x:ys),zs)
 | x > v  = (ys,(x:zs))
          where (ys,zs) = partition v xs
```

There are several heuristics for choosing `v`; let us choose it to be the first item of the given (nonempty) list. Here is the definition of `quicksort`.

```
quicksort [] = []
quicksort (x:xs) = (quicksort ys ) ++ [x] ++ (quicksort zs)
                 where (ys,zs) = partition x xs
```

Exercise 56

1. Show that each call in `quicksort` is made to a smaller argument.

2. What is the running time of `quicksort` if the input file is already sorted?

3. Find a permutation of 1 through 15 on which `quicksort` has the best performance; assume that the clause with guard `x <= v` executes slightly faster than the other clause.

   8 4 2 1 3 6 5 7 12 10 9 11 14 13 15

□

Exercise 57

1. Define a function that takes two lists of equal length as arguments and produces a boolean list of the same length as the result; an element of the boolean list is `True` iff the corresponding two elements of the argument lists are identical.

2. Define a function that creates a list of unique elements from a sorted list. Use this function to redefine function `uniq` of Section 5.9.4 (page 121).

3. Define function `zip` that takes a pair of lists of equal lengths as argument and returns a list of pairs of corresponding elements. So,

```
zip ([1,2,3], ['a','b','c']) = [(1,'a'), (2,'b'), (3,'c')]
```

4. Define function `unzip` that is the inverse of `zip`:

```
unzip (zip (xs,ys)) = (xs,ys)
```

5. Define function `take` where `take n xs` is a list containing the first `n` items of `xs` in order. If `n` exceeds the length of `xs` then the entire list `xs` is returned.

6. Define function `drop` where

```
xs = (take n xs) ++ (drop n xs)
```

7. Define function `index` where `index i xs` returns the `i`th element of `xs`. Assume that elements in a list are indexed starting at 0. Also, assume that the argument list is of length at least `i`. □

Exercise 58

A matrix can be represented as a list of lists. Let us adopt the convention that each outer list is a column of the matrix. Develop an algorithm to compute the determinant of a matrix of numbers. □

Exercise 59

It is required to develop a number of functions for processing an employee database. Each entry in the database has four fields: *employee*, *spouse*, *salary* and *manager*. The employee field is a string that is the name of the employee, the spouse field is the name of his/her spouse –henceforth, "his/her" will be abbreviated to "its" and "he/she" will be "it"—, the salary field is the employee's annual salary and the manager field is the name of employee's manager. Assume that the database contains all the records of a hierachical (tree-structured) organization in which every employee's spouse is also an employee, each manager is also an employee except *root*, who is the manager of all highest level managers. Assume that *root* does not appear as an employee in the database.

A manager of an employee is also called its *direct* manager; an *indirect* manager is either a direct manager or an indirect manager of a direct manager; thus, *root* is every employee's indirect manager.

Write functions for each of the following tasks. You will find it useful to define a number of auxiliary functions that you can use in the other functions. One such function could be *salary*, which given a name as an argument returns the corresponding salary.

In the following type expressions, DB is the type of the database, a list of 4-tuples, as described above.

1. Call an employee *overpaid* if its salary exceeds that of its manager. It is *grossly overpaid* if its salary exceeds the salaries of all its indirect managers. List all overpaid and grossly overpaid employees. Assume that the salary of *root* is 100,000.

```
overpaid         :: DB ->  [String]
grossly_overpaid :: DB ->  [String]
```

2. List all employees who directly manage their spouses; do the same for indirect management.

```
spouse_manager :: DB ->  [String]
```

3. List all managers who indirectly manage both an employee and its spouse.

```
indirect_manager :: DB ->  [String]
```

4. Are there employees e and f such that e's spouse is f's manager and f's spouse is e's manager?

```
nepotism :: DB ->  [(String,String)]
```

5. Find the family that makes the most money.

```
rich :: DB ->  [(String,String)]
```

6. Define the *rank* of a manager as the number of employees it manages. Define the *worth* of a manager as its salary/rank. Create three lists in which you list all managers in decreasing order of their salaries, ranks and worth.

```
sorted_salaries :: DB ->  [String]
sorted_rank     :: DB ->  [String]
sorted_worth    :: DB ->  [String]
```

7. The database is in *normal form* if the manager of x appears as an employee before x in the list. Write a function to convert a database to normal form. Are any of the functions associated with the above exercises easier to write or more efficient to run on a database that is given in normal form?

```
normalize :: DB -> DB
```

5.11 Higher Order Functions

Infix and Prefix Operators I use the term *operator* to mean a function of two arguments. An *infix operator*, such as `+`, is written between its two arguments, whereas a prefix operator, such as `max`, precedes its arguments. You can convert an infix operator to a prefix operator by putting parentheses around the function name (or symbol). Thus, `(+) x y` is the same as `x + y`. You can convert from prefix to infix by putting backquotes around an operator, so `div 5 3` is the same as `5 ‘div‘ 3`.

Most built-in binary operators in Haskell that do not begin with a letter, such as `+`, `*`, `&&`, and `||`, are infix; `max`, `min`, `rem`, `div`, and `mod` are prefix. □

5.11.1 Function `foldr`

We developed a number of functions —`suml`, `multl`— in Section 5.9.4 (page 118) that operate similarly on the argument list: (1) for the empty list, each function produces a specific value (0 for `suml`, 1 for `multl`) and (2) for a nonempty list, say `x:xs`, the item `x` and the function value for `xs` are combined using a specific operator (`+` for `suml`, `*` for `multl`). This suggests that we can code a generic function that has three arguments: the value supplied as in (1) —written as `z` below—, the function applied as in (2) — written as `f` below—, and the the list itself on which the function is to be applied. Here is such a function.

```
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Then we can define

```
suml  xs = foldr (+)  0  xs
multl xs = foldr (*)  1  xs
```

Similarly, we can define for boolean lists

```
andl  xs = foldr (&&) True   xs
orl   xs = foldr (||) False  xs
eql   xs = foldr (==) True   xs
```

The last one applies the equivalence operator (≡) over a list of booleans; the result is `True` if there are an even number of `False` elements in the list, and `False` otherwise.

We can define `flatten` of Section 5.10.1 (page 123) by

```
flatten xs = foldr (++) [] xs
```

Note I have been writing the specific operators, such as `(+)`, within parentheses, instead of writing them as just `+`, for instance, in the definition of `suml`. This is because the definition of foldr requires `f` to be a prefix operator, and `+` is an infix operator; `(+)` is the prefix version of `+`. □

Note There is an even nicer way to define functions such as `suml` and `multl`; just omit `xs` from both sides of the function definition. So, we have

```
suml  = foldr (+)  0
multl = foldr (*)  1
```

In these notes, I will not describe the justification for this type of definition. □

Function `foldr` has an argument that is a function; `foldr` is called a *higher order function*. The rules of Haskell do not restrict the type of argument of a function; hence, a function, being a typed value, may be supplied as an argument. Function (and procedure) arguments are rare in imperative programming, but they are common and very convenient to define and use in functional programming. Higher order functions can be defined for any type, not just lists.

What is the type of `foldr`? It has three arguments, `f`, `z` and `xs`, so its type is

(type of `f`) `->` (type of `z`) `->` (type of `xs`) `->` (type of result)

The type of `z` is arbitrary, say `a`. Then `f` takes two arguments of type `a` and produces a result of type `a`, so its type is `(a -> a -> a)`. Next, `xs` is a list of type `a`, so its type is `[a]`. Finally, the result type is `a`. So, we have for the type of `foldr`

```
(a -> a -> a) -> a -> [a] -> a
```

Actually, the interpreter gives a more general type:

```
Main> :t foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
```

This means that the two arguments of `f` need not be of the same type. Here is an example; function `evenl` determines if all integers of a given list are even. For its definition, we use function `ev` that takes an integer and a boolean as arguments and returns a boolean.

```
ev x  b   = (even x) && b
evenl xs  = foldr ev True xs

Main> evenl [10,20]
True
Main> evenl [1,2]
False
```

Function `fold` Function `fold` is a simpler version of `foldr`. It applies to nonempty lists only, and does not have the parameter `z`.

```
fold f [x] = x
fold f (x:xs) = f x (fold f xs)
```

Applying `fold` to the list `[a,b,c,d]` gives `f a (f b (f c d))`. To use an infix operator `!` on `[a,b,c,d]`, call `fold (!)[a,b,c,d]`. This gives `a!(b!(c!d))`, which is same as `a!b!c!d` when `!` is an associative operator. Quite often `fold` suffices in place of `foldr`. For example, we can define function `maxl` of Section 5.9.4, which computes the maximum element of a nonempty list, by

```
maxl = fold max
```

5.11.2 Function `map`

Function `map` takes as arguments (1) a function `f` and (2) a list of elements on which `f` can be applied. It returns the list obtained by applying `f` to each element of the given list.

```
map f [] = []
map f (x:xs) = (f x) : (map f xs)
```

So,

```
Main> map not [True,False]
[False,True]
Main> map even [2,4,5]
[True,True,False]
Main> map chCase "jmisra"
"JMISRA"
Main> map len ["Jayadev","Misra"]
[7,5]
```

The type of map is:

```
Main> :t map
map :: (a -> b) -> [a] -> [b]
```

This function is so handy that it is often used to transform a list to a form that can be more easily manipulated. For example, to determine if all integers in a given list, `xs`, are even, we write

```
andl (map even xs)
```

where `andl` is defined in Section 5.11.1 (page 133). Here `(map even xs)` creates a list of booleans of the same length as the list `xs` such that the ith boolean is `True` iff the ith element of `xs` is even. The function `andl` then take the conjunction of the booleans in this list.

Exercise 60

Redefine the functions `cons0` and `cons1` from page 127 using `map`. □

5.11.3 Function `filter`

Function `filter` has two arguments, a predicate `p` and a list `xs`; it returns the list containing the elements of `xs` for which `p` holds.

```
filter p []  = []
filter p (x:xs)
 | p x       = x: (filter p xs)
 | otherwise = (filter p xs)
```

So, we have

```
Main> filter even [2,3,4]
[2,4]
Main> filter digit ['a','9','b','0','c']
"90"
Main> filter upper "Jayadev Misra"
"JM"
Main> filter digit "Jayadev Misra"
""
```

The type of `filter` is

```
Main> :t filter
filter :: (a -> Bool) -> [a] -> [a]
```

Exercise 61

What is `filter p (filter q xs)`? In particular, what is `filter p (filter (not p) xs)`? □

5.12 Program Design: Boolean Satisfiability

We treat a longer example —boolean satisfiability— in this section. The problem is to determine if a propositional boolean formula is *satisfiable*, i.e., if there is an assignment of (boolean) values to variables in the formula that makes the formula *true*. For example, $(p \vee q) \wedge (\neg p \vee \neg q) \wedge (p \vee \neg q)$ is satisfiable with $p, q = true,\ false$.

In Section 5.12.1, I introduce the problem more precisely and present the Davis-Putnam procedure, which is an effective solution method for this problem. In Section 5.12.2, I develop a Haskell implementation of this procedure by choosing a suitable data structure and then presenting an appropriate set of functions in a top-down fashion.

5.12.1 Boolean Satisfiability

The satisfiability problem is an important problem in computer science, one that has surprising applicability to a diverse range of problems, such as circuit design, theorem proving and robotics. It has been studied since the early days of computing science. Indeed, a landmark theoretical paper by Steve Cook demonstrated that if someone could devise a fast algorithm for the general satisfiability problem, it could be used to solve many other difficult problems quickly. As a result, for a long time, satisfiability was considered too difficult to tackle, and a lot of effort was invested in trying to design domain-specific heuristics to solve specific instances of satisfiability.

However, recent advances in the design of "satisfiability solvers" (algorithms for solving instances of the satisfiability problem) have changed this perception. Although the problem remains difficult in general, recent satisfiability solvers, such as Chaff [36], employ clever data structures and learning techniques that prove to work surprisingly well in practice (for reasons no one quite understands).

Today, satisfiability solvers are used as the core engine in a variety of industrial products. CAD companies like Synopsys and Cadence use them as the engine for their tools for property checking and microprocessor verification [33], in automatic test pattern generation [30], and even in FPGA routing [16]. Verification engineers at Intel, Motorola and AMD incorporate satisfiability solvers in their tools for verifying their chip designs. Researchers in artificial intelligence and robotics are discovering that their planning problems can be cast as boolean satisfiability, and solvers like Chaff outperform even their domain-specific planning procedures [24]. Other researchers are increasingly beginning to use satisfiability solvers as the engine inside their model checkers, theorem provers, program checkers, and even optimizing compilers.

Conjunctive Normal Form

A propositional formula is said to be in *Conjunctive Normal Form* (*CNF*) if it is the conjunction of a number of *terms*, where each term is the disjunction of a number of *literals*, and each literal is either a variable or its negation. For example, the formula $(p \vee q) \ \wedge \ (\neg p \vee \neg q) \ \wedge \ (p \vee \neg q)$ is in CNF. Any boolean formula can be converted to a logically equivalent CNF formula. Henceforth, we assume that the input formula is in CNF.

Complexity of the Satisfiability Problem

The satisfiability problem has been studied for over five decades. It may seem that the problem, particularly the CNF version, is easy to solve: each term has to be made *true* for the entire conjunction to be *true* and a term can be made *true* by making any constituent literal *true*. However, the choice of literal for one term may conflict with another: for $(p \vee q) \ \wedge \ (\neg p \vee \neg q)$, if p is chosen to be *true* for the first term and $\neg p$ is chosen to be *true* for the second term, there is a conflict.

There is no known polynomial algorithm for the satisfiability problem. In fact, the CNF satisfiability problem in which each term has exactly 3 literals — known as 3-SAT— is NP-complete, though 2-SAT can be solved in linear time. However, there are several solvers that do extremely well in practice. The Chaff solver [47] can determine satisfiability of a formula with hundreds of thousands of variables and over a million terms in an hour, or so, on a PC circa 2002. This astounding speed can be attributed to (1) a very good algorithm, the Davis-Putnam procedure, which we study next, (2) excellent heuristics, and (3) fast computers with massive main memories.

The Davis-Putnam procedure

To explain the procedure, I will use the following formula, f, over variables p, q and r:

$$f \;::\; (\neg p \vee q \vee r) \;\wedge\; (p \vee \neg r) \;\wedge\; (\neg q \vee r) \;\wedge\; (\neg p \vee \neg q \vee \neg r) \;\wedge\; (p \vee q \vee r)$$

Next, we ask whether f is satisfiable, *given that p is true*. Then, any term in f that contains p becomes *true*, and can be removed from consideration (since it is part of a conjunction). Any term that contains $\neg p$, say $(\neg p \vee q \vee r)$, can become *true* only by making $(q \vee r)$ *true* . Therefore, given that p is *true*, f is satisfiable iff f_p is satisfiable, where

$$f_p \;::\; (q \vee r) \;\wedge\; (\neg q \vee r) \;\wedge\; (\neg q \vee \neg r)$$

Note that p does not appear in f_p.

Similarly, given that p is *false*, f is satisfiable provided that $f_{\neg p}$ is satisfiable, where

$$f_{\neg p} \;::\; (\neg r) \;\wedge\; (\neg q \vee r) \;\wedge\; (q \vee r)$$

We have two mutually exclusive possibilities: p is *true* and $\neg p$ is *true* . Therefore, f is satisfiable iff either f_p is satisfiable or $f_{\neg p}$ is satisfiable. Thus, we have divided the problem into two smaller subproblems, each of which may be decomposed further in a similar manner. Ultimately, we will find that

- a formula is empty (i.e., it has no terms), in which case it is satisfiable, because it is a conjunction, or
- some term in the formula is empty (i.e., it has no literals), in which case the term (and hence also the formula) is unsatisfiable, because it is a disjunction.

As long as neither possibility holds, we have a nonempty formula none of whose terms is empty, and we can continue with the decomposition process.

The entire procedure can be depicted as a binary tree, see Figure 5.2, where each node has an associated formula (whose satisfiability is being computed) and each edge has the name of a literal. The literals on the two outgoing edges from a node are negations of each other. The leaf nodes are marked either `F`

or `T`, corresponding to *false* and *true*. A path in the tree corresponds to an assignment of values to the variables. The value, `F` or `T`, at a leaf is the value of the formula (which is associated with the root) for the variable values assigned along the path to the leaf. Thus, in Figure 5.2, assigning *true* to p, $\neg q$ and r makes f *true* (therefore, f is satisfiable). For all other assignments of variable values, f is *false*.

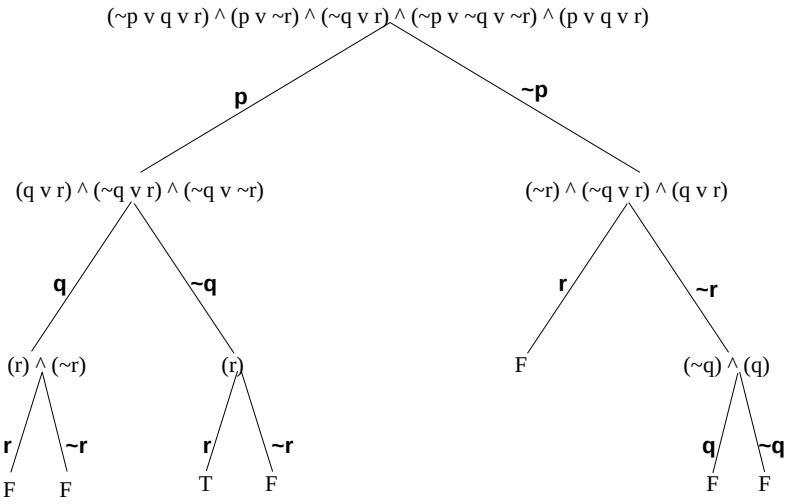


Figure 5.2: Davis-Putnam Computation

Note Assume that no term contains a variable and its negation.

5.12.2 Program Development

Data Structure Here, we consider representation of a formula in Haskell. Recall that

> a *formula* is a conjunction of a set of *terms*.
> a *term* is the disjunction of a set of *literals*, and
> a *literal* is either a variable or the negation of a variable.

We can regard a formula as a list of terms and a term as a list of literals; conjunction and disjunction are implicit. Now, we consider representations of literals. A variable may be represented as a string, and a negation, such as $\neg p$, by a string, `"-p"`, whose first symbol is `"-"`. To preserve symmetry, let me add a `"+"` sign in front of a variable; so p is represented by `"+p"`. Then the representation of f is:

```
[
 ["-p", "+q", "+r"],
 ["+p", "-r"],
 ["-q", "+r"],
 ["-p", "-q", "-r"],
 ["+p", "+q", "+r"]
]
```

We should be clear about the types. I define the types explicitly (I have not told you how to do this in Haskell; just take it at face value).

```
type Literal = String
type Term    = [Literal]
type Formula = [Term]
```

The top-level function I define a function `dp` that accepts a formula as input and returns a boolean, `True` if the formula is satisfiable and `False` otherwise. So, we have

```
dp :: Formula -> Bool
```

If the formula is empty, then the result is `True`. If it contains an empty term, then the result is `False`. Otherwise, we choose some literal of the formula, decompose the formula into two subformulae based on this literal, solve each subformula recursively for satisfiability, and return `True` if either returns `True`.

```
dp xss
 | xss == []   = True
 | emptyin xss = False
 | otherwise   = (dp yss) || (dp zss)
                 where
                  v = literal xss
                  yss = reduce v xss
                  zss = reduce (neg v) xss
```

We have introduced the following functions which will be developed next:

`emptyin xss`: returns `True` if `xss` has an empty term,

`literal xss`: returns a literal from `xss`, where `xss` is nonempty and does not contain an empty term,

`neg v`: returns the string corresponding to the negation of `v`.

`reduce v xss`, where `v` is a literal: returns the formula obtained from `xss` by dropping any term containing the literal `v` and dropping any occurrence of the literal `neg v` in each remaining term.

Functions `emptyin`, `literal`, `reduce`, `neg`

The code for `emptyin` is straightforward:

```
-- Does the formula contain an empty list?
emptyin :: Formula -> Bool
emptyin []          = False
emptyin ([]: xss)   = True
emptyin (xs: xss)   = emptyin xss
```

Function `literal` can return any literal from the formula; it is easy to return the first literal of the first term of its argument. Since the formula is not empty and has no empty term, this procedure is valid.

```
{- Returns a literal from a formula.
   It returns the first literal of the first list.
   The list is not empty, and
   it does not contain an empty list.
-}
literal :: Formula -> Literal
literal ((x: xs):xss) = x
```

A call to `reduce v xss` scans through the terms of `xss`. If `xss` is empty, the result is the empty formula. Otherwise, for each term `xs`,

- if `v` appears in `xs`, drop the term,
- if the negation of `v` appears in `xs` then modify `xs` by removing the negation of `v`,
- if neither of the above conditions hold, retain the term.

```
-- reduce a literal through a formula
reduce :: Literal -> Formula -> Formula
reduce v []         = []
reduce v (xs:xss)
 | inl v xs       = reduce v xss
 | inl (neg v) xs = (remove (neg v) xs): (reduce v xss)
 | otherwise      = xs : (reduce v xss)
```

Finally, `neg` is easy to code:

```
-- negate a literal
neg :: Literal -> Literal
neg ('+': var) = '-': var
neg ('-': var) = '+': var
```

Function `reduce` introduces two new functions, which will be developed next.

`inl v xs`, where `v` is a literal and `xs` is a term: returns `True` iff `v` appears in `xs`,

`remove u xs`, where `u` is a literal known to be in term `xs`: return the term obtained by removing `u` from `xs`.

Functions `inl`, `remove`

Codes for both of these functions are straightforward:

```
-- check if a literal is in a term
inl :: Literal -> Term -> Bool
inl v [] = False
inl v (x:xs) = (v == x) || (inl v xs)

-- remove a literal u from term xs. u is in xs.
remove :: Literal -> Term -> Term
remove u (x:xs)
 | x == u    = xs
 | otherwise = (x : (remove u xs))
```

Exercise 62

1. Test the program.
2. Function `reduce` checks if `xss` is empty. Is this necessary given that `reduce` is called from `dp` with a nonempty argument list? Why doesn't `reduce` check whether `xss` has an empty term?
3. Rewrite `dp` so that if the formula is satisfiable, it returns the assignments to variables that make the formula *true*. □

5.12.3 Variable Ordering

It is time to take another look at our functions to see if we can improve any of them, either the program structure or the performance. Actually, we can do both.

Note that in `reduce` we look for a literal by scanning all the literals in each term. What if we impose an order on the variables and write each term in the given order of variables? Use the following ordering in f: `[ "p", "q" ,"r" ]`. Then, a term like $(\neg p \vee q \vee r)$ is ordered whereas $(\neg p \vee r \vee q)$ is not. If each term in the formula is ordered and in `reduce v xss`, `v` is the *smallest* literal in `xss`, then we can check the first literal of each term to see whether it contains `v` or its negation.

Function `dp2`, given below, does the job of `dp`, but it needs an extra argument, a list of variables like `[ "p", "q" ,"r" ]`, which defines the variable ordering. Here, `reduce2` is the counterpart of `reduce`. Now we no longer need the functions `inl`, `remove` and `literal`.

```
dp2 vlist xss
 | xss == []   = True
 | emptyin xss = False
 | otherwise   = (dp2 wlist yss) || (dp2 wlist zss)
                 where
                  v:wlist = vlist
                  yss     = reduce2 ('+': v) xss
                  zss     = reduce2 ('-': v) xss

-- reduce a literal through a formula
reduce2 :: Literal -> Formula -> Formula
reduce2 w []         = []
reduce2 w ((x:xs):xss)
 | w == x       = reduce2 w xss
 | (neg w) == x = xs: (reduce2 w xss)
 | otherwise    = (x:xs): (reduce2 w xss)
```

A further improvement is possible. Note that `reduce2` scans its argument list twice, once for `w` and again for `neg w`. We define `reduce3` to scan the given list only once, to create two lists, one in which `w` is removed and the other in which `neg w` is removed. Such a solution is shown below, where `reduce3` is the counterpart of `reduce2`. Note that the interface to `reduce3` is slightly different. Also, we have eliminated the use of function `neg`.

```
dp3 vlist xss
 | xss == []   = True
 | emptyin xss = False
 | otherwise   = (dp3 wlist yss) || (dp3 wlist zss)
                 where
                  v:wlist   = vlist
                  (yss,zss) = reduce3 v xss

reduce3 v []         = ([],[])
reduce3 v ((x:xs):xss)
 | '+': v == x = (yss   , xs:zss)
 | '-': v == x = (xs:yss, zss   )
 | otherwise   = ((x:xs):yss, (x:xs):zss)
                   where
                    (yss,zss) = reduce3 v xss
```

Exercise 63

1. What are the types of `dp2`, `reduce2`, `dp3` and `reduce3`?
2. When `dp2 vlist xss` is called initially, `vlist` is the list of names of the variables in `xss`. Argue that the program maintains this as an invariant. In particular, `vlist` is empty iff `xss` is empty.

3. Is there any easy way to eliminate the function `emptyin`? In particular, can we assert that any empty term will be the first term in a formula?

4. Here is another strategy that simplifies the program structure and improves the performance. Convert each variable to a distinct positive integer (and its negation to the corresponding negative value). Make sure that each term is an increasing list (in magnitude). Having done this, the argument `vlist` is no longer necessary. Modify the solution to accommodate these changes. □

Acknowledgement I am grateful to Ham Richards and Kaustubh Wagle for thorough readings of these notes, and many suggestions for improvements. Rajeev Joshi has provided me with a number of pointers to the applications of boolean satisfiability.

Chapter 6

Relational Database

6.1 Introduction

You can now purchase a music player that stores nearly 10,000 songs. The storage medium is a tiny hard disk, a marvel of hardware engineering. Equally impressive is the software which combines many aspects of compression, error correction and detection, and database manipulation.

First, the compression algorithm manages to store around 300 music CDs, each with around 600MB of storage, on my 20GB player; this is a compression of about 10 to 1. While it is possible to compress music to any extent, because exact reproduction is not expected, you would not want to listen to such music. Try listening to a particularly delicate piece over the telephone! The compression algorithm manages to reproduce music reasonably faithfully.

A music player begins its life expecting harsh treatment, even torture. The devices are routinely dropped, they are subjected to X-ray scans at airports, and left outside in very cold or very hot cars. Yet, the hardware is reasonably resilient, but more impressively, the software works around the hardware glitches using error-correcting strategies some of which we have outlined in an earlier chapter.

The question that concerns us in this chapter is how to *organize* a large number of songs so that we can locate a set of songs quickly. The songs are first stored on a desktop (being imported from a CD or over the internet from a music store); they can be organized there and then downloaded to a player. A naive organization will make it quite frustrating to find that exact song in your player. And, you may wish to listen to all songs which are either by artist A or composer B, in the classical genre, and have not been played more than 6 times in the last 3 months. The subject matter of this chapter is organization of certain kinds of data, like songs, to allow efficient selection of a subset which meets a given search criterion.

For many database applications a set of tuples, called a *table*, is often the appropriate data structure. Let me illustrate it with a small database of movies;

Title	Actor	Director	Genre	Year
Jurassic Park	Jeff Goldblum	Steven Spielberg	Action	1993
Jurassic Park	Sam Neill	Steven Spielberg	Action	1993
Men in Black	Tommy Lee Jones	Barry Sonnenfeld	SciFi	1997
Men in Black	Will Smith	Barry Sonnenfeld	SciFi	1997
Independence Day	Will Smith	Roland Emmerich	SciFi	1996
Independence Day	Bill Pullman	Roland Emmerich	SciFi	1996
My Fair Lady	Audrey Hepburn	George Cukor	Classics	1964
My Fair Lady	Rex Harrison	George Cukor	Classics	1964
The Sound of Music	Julie Andrews	Robert Wise	Classics	1965
The Sound of Music	Christopher Plummer	Robert Wise	Classics	1965
Bad Boys II	Martin Lawrence	Michael Bay	Action	2003
Bad Boys II	Will Smith	Michael Bay	Action	2003
Ghostbusters	Bill Murray	Ivan Reitman	Comedy	1984
Ghostbusters	Dan Aykroyd	Ivan Reitman	Comedy	1984
Tootsie	Dustin Hoffman	Sydney Pollack	Comedy	1982
Tootsie	Jessica Lange	Sydney Pollack	Comedy	1982

Table 6.1: A list of movies arranged in a table

Title	Actor	Director	Genre	Year
Men in Black	Will Smith	Barry Sonnenfeld	SciFi	1997
Independence Day	Will Smith	Roland Emmerich	SciFi	1996

Table 6.2: Result of selection on Table 6.1 (page 146)

see Table 6.1 (page 146). We store the following information for each movie: its title, actor, director, genre and the year of release. We list only the two most prominent actors for a movie, and they have to appear in different tuples; so each movie is being represented by two tuples in the table. We can now easily specify a search criterion such as, find all movies released between 1980 and 2003 in which Will Smith was an actor and the genre is SciFi. The result of this search is a table, shown in Table 6.2 (page 146).

Chapter Outline We introduce the table data structure and some terminology in section 6.2. A table resembles a mathematical relation, though there are some significant differences which we outline in that section. An algbra of relations is developed in section 6.3. The algebra consists of a set of operations on relations (section 6.3.1) and a set of identities over relational expressions (section 6.3.2). The identities are used to process queries efficiciently, as shown in section 6.3.3. A standard query language, SQL, is described in section 6.4. This chapter is a very short introduction to the topic; for more thorough treatment see the relevant chapters in [32] and [2].

6.2 The Relational Data Model

Central to the relational data model is the concept of *relation*. You are familiar with relations from algebra, which I briefly review below. Next, I will explain relations in databases, which are slightly different.

6.2.1 Relations in Mathematics

The $>$ operator over positive integers is a (binary) relation. We write $5 > 3$, using the relation as an infix operator. More formally, the relation $>$ is a set of pairs:

$$\{(2,1),\ (3,1),\ (3,2),\ \cdots\}$$

A general relation consists of tuples, not necessarily pairs as for binary relations. Consider a *family* relation which consists of triples (c, f, m), where c is the name of a child, and f and m are the father and the mother. Or, the relation *Euclid* which consists of triples (x, y, z) where the components are positive integers and $x^2 + y^2 = z^2$. Or, *Fermat* which consists of quadruples of positive integers (x, y, z, n), where $x^n + y^n = z^n$ and $n > 2$. (A recent breakthrough in mathematics has established that $Fermat = \phi$.) In databases, the relations need not be binary; in fact, most often, they are not binary.

A relation, being a set, has all the set operations defined on it. We list some of the set operations below which are used in relational algebra.

1. Union: $R \cup S \ = \ \{x |\ x \in R \ \vee \ x \in S\}$
2. Intersection: $R \cap S \ = \ \{x |\ x \in R \ \wedge \ x \in S\}$
3. Difference: $R - S \ = \ \{x |\ x \in R \ \vee \ x \notin S\}$
4. Cartesian Product: $R \times S \ = \ \{(x, y) |\ x \in R \ \wedge \ y \in S\}$

Thus, given $R \ = \ \{(1,2),(2,3),(3,4)\}$ and $S \ = \ \{(2,3),(3,4),(4,5)\}$, we get

$$\begin{array}{rcl} R \cup S & = & \{(1,2),(2,3),(3,4),(4,5)\} \\ R \cap S & = & \{(2,3),(3,4)\} \\ R - S & = & \{(1,2)\} \\ R \times S & = & \{((1,2),(2,3)),((1,2),(3,4)),((1,2),(4,5)), \\ & & \ ((2,3),(2,3)),((2,3),(3,4)),((2,3),(4,5)), \\ & & \ ((3,4),(2,3)),((3,4),(3,4)),((3,4),(4,5))\} \end{array}$$

In algebra, you have seen *reflexive*, *symmetric*, *asymmetric* and *transitive* binary relations. None of these concepts is of any use in relational algebra.

Year	Genre	Title	Director	Actor
1997	SciFi	Men in Black	Barry Sonnenfeld	Will Smith
1996	SciFi	Independence Day	Roland Emmerich	Will Smith

Table 6.3: A column permutation of Table 6.2 (page 146)

Theatre	Address
General Cinema	2901 S 360
Tinseltown USA	5501 S I.H. 35
Dobie Theater	2021 Guadalupe St
Entertainment Film	6700 Middle Fiskville Rd

Table 6.4: Theatres and their addresses

6.2.2 Relations in Databases

Database relations are inspired by mathematical relations. A database relation is best represented by a matrix, called a *table*, in which (1) each row is a tuple and (2) each column has a name, which is an *attribute* of the relation. Table 6.1 (page 146) shows such a relation; it has 5 attributes: Title, Actor, Director, Genre, Year. There are 16 rows, each is a tuple of the relation.

In both mathematical and database relations, the tuples are distinct and they may appear in any order. The type of an attribute, i.e., the type of values that may appear in that column, is called the *domain* of the attribute. The name of a database relation along with the names and domains of attributes is called a *relational schema*. A schema is a template; an instance of the schema has a number of tuples which fit the template.

The most fundamental difference between mathematical and database relations is that in the latter the columns can be permuted arbitrarily keeping the same relation. Thus, Table 6.2 (page 146) and Table 6.3 (page 148) represent the same relation. Therefore, we have the identity (we explain $R \times S$, the cartesian product of database relations R and S, in section 6.3.1).

$$R \times S \;=\; S \times R$$

For mathematical relations, this identity does not hold because the components cannot be permuted.

A relational database is a set of relations with distinct relation names. The relations in Tables 6.1 (page 146), 6.4 (page 148), and 6.5 (page 149) make up a relational database. Typically, every relation in a database has a common attribute with some other relation.

Theatre	Title	Time	Rating
General Cinema	Jurassic Park	Sat, 9PM	G
General Cinema	Men in Black	Sat, 9PM	PG
General Cinema	Men in Black	Sun, 3PM	PG
Tinseltown USA	Independence Day	Sat, 9PM	PG-13
Dobie Theater	My Fair Lady	Sun, 3PM	G
Entertainment Film	Ghostbusters	Sun, 3PM	PG-13

Table 6.5: Theatres, Movies, Time and Rating

6.3 Relational Algebra

An algebra consists of (1) elements, (2) operations and (3) identities. For example, to do basic arithmetic over integers we define: (1) elements to be integers, (2) operations to be $+$, $-$, $\times$, $\div$, and (3) identities such as,

$$x + y = y + x$$
$$x \times (y + z) = x \times y + x \times z$$

where x, y and z range over the elements (i.e., integers).

We define an algebra of database relations in this section. The elements are database relations. We define a number of operations on them in section 6.3.1 and several identities in section 6.3.2.

6.3.1 Operations on Database Relations

Henceforth, R, S and T denote relations, and a and b are sets of attributes. Relations R and S are *union-compatible*, or just *compatible*, if they have the same set of attributes.

Union $R \cup S$ is the union of *compatible* relations R and S. Relation $R \cup S$ includes all tuples from R and S with duplicates removed.

Intersection $R \cap S$ is the intersection of *compatible* relations R and S. Relation $R \cap S$ includes all tuples which occur in both R and S.

Difference $R-S$ is the set difference of *compatible* relations R and S. Relation $R - S$ includes all tuples which are in R and not in S.

Cartesian Product or Cross Product $R \times S$ is the cross product of relations R and S. The relations need not be compatible. Assume for the moment that the attributes of R and S are disjoint. The set of attributes of $R \times S$ are the ones from both R and S. Each tuple of R is concatenated with each tuple of S to form tuples of $R \times S$. Two database relations are shown in Table 6.6

Title	Actor	Director	Year
Jurassic Park	Sam Neill	Steven Spielberg	1993
Men in Black	Tommy Lee Jones	Michael Bay	2003
		Ivan Reitman	1984

Table 6.6: Two relations separated by vertical line

Title	Actor	Director	Year
Jurassic Park	Sam Neill	Steven Spielberg	1993
Jurassic Park	Sam Neill	Michael Bay	2003
Jurassic Park	Sam Neill	Ivan Reitman	1984
Men in Black	Tommy Lee Jones	Steven Spielberg	1993
Men in Black	Tommy Lee Jones	Michael Bay	2003
Men in Black	Tommy Lee Jones	Ivan Reitman	1984

Table 6.7: Cross Product of the two relations in Table 6.6 (page 150)

(page 150); they are separated by a vertical line. Their cross product is shown in Table 6.7 (page 150).

The cross product in Table 6.7 makes no sense. We introduce the *join* operator later in this section which takes a more "intelligent" cross product.

If R and S have common attribute names, the names are changed so that we have disjoint attributes. One strategy is to prefix the attribute name by the name of the relation. So, if you are computing *Prof* $\times$ *Student* where both *Prof* and *Student* have an attribute *id*, an automatic renaming may create *Profid* and *Studentid*. This does not always work, for instance, in *Prof* $\times$ *Prof*. Manual aid is then needed. In this chapter, we write $R \times S$ only if the attributes of R and S are disjoint.

Note a subtle difference between mathematical and database relations for cross product. For tuple (r, s) in R and (u, v) in S, their mathematical cross product gives a tuple of tuples, $((r, s), (u, v))$, whereas the database cross product gives a tuple containing all 4 elements, (r, s, u, v).

The number of tuples in $R \times S$ is the number of tuples in R times the number in S. Thus, if R and S have 1,000 tuples each, $R \times S$ has a million tuples and $R \times (S \times S)$ has a billion. So, cross product is rarely computed in full. It is often used in conjunction with other operations which can be applied in a clever sequence to eliminate explicit computations required for a cross product.

Projection The operations we have described so far affect only the rows (tuples) of a table. The next operation, *projection*, specifies a set of attributes of a relation that are to be retained to form a relation. Projection removes all other attributes (columns), and removes any duplicate rows that are created as a result. We write $\pi_{u,v}(R)$ to denote the relation which results by retaining only the attributes u and v of R. Let R be the relation shown in Table 6.1

(page 146). Then, $\pi_{Title,Director,Genre,Year}(R)$ gives Table 6.9 (page 152) and $\pi_{Title,Actor}(R)$ gives Table 6.10 (page 153).

Selection The selection operation chooses the tuples of a relation that satisfy a specified predicate. A predicate uses attribute names as variables, as in $year \geq 1980 \wedge year \leq 2003 \wedge actor = \text{“}Will\ Smith\text{”} \wedge genre = \text{“}SciFi\text{”}$. A tuple satisfies a predicate if the predicate is *true* when the attribute names are replaced by the corresponding values from the tuple. We write $\sigma_p(R)$ to denote the relation consisting of the subset of tuples of R that satisfy predicate p. Let R be the relation in Table 6.1 (page 146). Then, $\sigma_{year \geq 1980 \wedge year \leq 2003 \wedge actor = \text{“}Will\ Smith\text{”} \wedge genre = \text{“}SciFi\text{”}}(R)$ is shown in Table 6.2 (page 146) and $\sigma_{actor = \text{“}Will\ Smith\text{”} \wedge genre = \text{“}Comedy\text{”}}(R)$ is the empty relation.

Join There are several *join* operators in relational algebra. We study only one which is called *natural join*, though we simply call it *join* in this chapter. The join of R and S is written as $R \bowtie S$. Here, R and S need not be compatible; typically, they will have some common attributes.

The join is a more refined way of taking the cross product. As in the cross product, take each tuple r of R and s of S. If r and s match in their common attributes, concatenate them keeping only one set of columns for the common attributes. Consider Tables 6.4 (page 148) and 6.5 (page 149). Their join is shown in Table 6.8 (page 151). And, the join of Tables 6.9 (page 152) and 6.10 (page 153) is Table 6.1 (page 146).

Theatre	Title	Time	Rating	Address
General Cinema	Jurassic Park	Sat, 9PM	G	2901 S 360
General Cinema	Men in Black	Sat, 9PM	PG	2901 S 360
General Cinema	Men in Black	Sun, 3PM	PG	2901 S 360
Tinseltown USA	Independence Day	Sat, 9PM	PG-13	5501 S I.H. 35
Dobie Theater	My Fair Lady	Sun, 3PM	G	2021 Guadalupe St
Entertainment Film	Ghostbusters	Sun, 3PM	PG-13	6700 Middle Fiskville

Table 6.8: Join of Tables 6.4 and 6.5

If R and S have no common attributes, $R \bowtie S$ is an empty relation, though it has all the attributes of R and S. We will avoid taking $R \bowtie S$ if R and S have no common attributes.

Writing $attr(R)$ for the set of attributes of R, we have

$$attr(R \bowtie S) = attr(R) \cup attr(S), \text{ and}$$
$$x \in R \bowtie S \equiv (attr(R) \cap attr(S) \neq \phi) \wedge \pi_{attr(R)}(x) \in R \wedge \pi_{attr(S)}(x) \in S$$

The condition $attr(R) \cap attr(S) \neq \phi$, i.e., R and S have a common attribute, is essential. Without this condition, $R \bowtie S$ would be $R \times S$ in case the attributes are disjoint.

Title	Director	Genre	Year
Jurassic Park	Steven Spielberg	Action	1993
Men in Black	Barry Sonnenfeld	SciFi	1997
Independence Day	Roland Emmerich	SciFi	1996
My Fair Lady	George Cukor	Classics	1964
The Sound of Music	Robert Wise	Classics	1965
Bad Boys II	Michael Bay	Action	2003
Ghostbusters	Ivan Reitman	Comedy	1984
Tootsie	Sydney Pollack	Comedy	1982

Table 6.9: Compact representation of a portion of Table 6.1 (page 146)

The join operator selects only the tuples which match in certain attributes; so, join results in a much smaller table than the cross product. Additionally, the result is usually more meaningful. In many cases, a large table can be decomposed into two much smaller tables whose join recreates the original table. See the relations in Tables 6.9 (page 152) and 6.10 (page 153) whose join gives us the relation in Table 6.1. The storage required for these two relations is much smaller than that for Table 6.1 (page 146).

Title	Actor
Jurassic Park	Jeff Goldblum
Jurassic Park	Sam Neill
Men in Black	Tommy Lee Jones
Men in Black	Will Smith
Independence Day	Will Smith
Independence Day	Bill Pullman
My Fair Lady	Audrey Hepburn
My Fair Lady	Rex Harrison
The Sound of Music	Julie Andrews
The Sound of Music	Christopher Plummer
Bad Boys II	Martin Lawrence
Bad Boys II	Will Smith
Ghostbusters	Bill Murray
Ghostbusters	Dan Aykroyd
Tootsie	Dustin Hoffman
Tootsie	Jessica Lange

Table 6.10: Table 6.1 (page 146) arranged by Title and Actor

Exercise 64
Suppose $R \times S$ is defined. What is $R \bowtie S$?

Exercise 65
Suppose R and S are compatible. Show that $R \bowtie S \ = \ R \cap S$.

6.3.2 Identities of Relational Algebra

We develop a number of identities in this section. I don't prove the identities; I recommend that you do. These identities are used to transform a relational expression into an equivalent form whose evaluation is more efficicient, a procedure known as query optimization. Query optimization can reduce evaluation time of relational expressions by several orders of magnitude. In the following, R , S and T denote relations, a and b are sets of attributes, and p and q are predicates.

1. (Selection splitting) $\sigma_{p \wedge q}(R) \ = \ \sigma_p(\sigma_q(R))$
2. (Commutativity of selection)

   $$\sigma_p(\sigma_q(R)) \ = \ \sigma_q(\sigma_p(R))$$

   This is a corollary of Selection splitting given above.
3. (Projection refinement) Let a and b be subsets of attributes of relation R, and $a \subseteq b$. Then,

$$\pi_a(R) \;=\; \pi_a(\pi_b(R))$$

4. (Commutativity of selection and projection) $\pi_a(\sigma_p(R)) \;=\; \sigma_p(\pi_a(R))$

5. (Commutativity and Associativity of union, cross product, join)

$$\begin{aligned}
&R \cup S \;=\; S \cup R \\
&(R \cup S) \cup T \;=\; R \cup (S \cup T) \\
&R \times S \;=\; S \times R \\
&(R \times S) \times T \;=\; R \times (S \times T) \\
&R \bowtie S \;=\; S \bowtie R \\
&(R \bowtie S) \bowtie T \;=\; R \bowtie (S \bowtie T),
\end{aligned}$$

   provided R and S have common attributes and so do S and T, and no attribute is common to all three relations.

6. (Selection pushing)

$$\begin{aligned}
\sigma_p(R \cup S) \;&=\; \sigma_p(R) \cup \sigma_p(S) \\
\sigma_p(R \cap S) \;&=\; \sigma_p(R) \cap \sigma_p(S) \\
\sigma_p(R - S) \;&=\; \sigma_p(R) - \sigma_p(S)
\end{aligned}$$

   Suppose predicate p names only attributes of R. Then,

$$\begin{aligned}
\sigma_p(R \times S) \;&=\; \sigma_p(R) \times S \\
\sigma_p(R \bowtie S) \;&=\; \sigma_p(R) \bowtie S
\end{aligned}$$

7. (Projection pushing)

$$\pi_a(R \cup S) \;=\; \pi_a(R) \cup \pi_a(S)$$

8. (Distributivity of projection over join)

$$\pi_a(R \bowtie S) \;=\; \pi_a(\pi_b(R) \bowtie \pi_c(S))$$

   where R and S have common attributes d, a is a subset of attributes of both R and S, b is a's subset from R plus d and c is a's subset from S plus d. That is,

$$\begin{aligned}
&a \subseteq attr(R) \;\cup\; attr(S) \\
&b = (a \cap attr(R)) \;\cup\; d \\
&c = (a \cap attr(S)) \;\cup\; d \\
&d = attr(R) \;\cap\; attr(S)
\end{aligned}$$

□

Selection splitting law says that evaluations of $\sigma_{p \wedge q}(R)$ and $\sigma_p(\sigma_q(R))$ are interchangeable; so, apply either of the following procedures: look at each tuple of R and decide if it satisfies $p \wedge q$, or first identify the tuples of R which satisfy q and from those identify the ones which satisfy p. The benefit of one strategy over another depends on the relative costs of access times to the tuples and predicate evaluation times. For large databases, which are stored in secondary storage, access time is the major cost. Then it is preferable to evaluate $\sigma_{p \wedge q}(R)$.

The distributivity of selection over join is often used in query optimization. It is a good heuristic to apply selection to as small a relation as possible. Therefore, it is almost always better to evluate $\sigma_p(R) \bowtie S$ instead of $\sigma_p(R \bowtie S)$, i.e., apply selection to R which tends to be smaller than $R \bowtie S$.

Exercise 66

Suppose predicate p names only the attributes of S. Show that $\sigma_p(R \bowtie S) = R \bowtie \sigma_p(S)$.

Exercise 67

Show that $\pi_a(R \cap S) = \pi_a(R) \cap \pi_a(S)$ does not necessarily hold.

6.3.3 Example of Query Optimization

We consider the relations in Tables 6.1 (page 146), 6.5 (page 149), and 6.4 (page 148). We call these relations R, S and T, respectively. Relation R is prepared by some movie distribution agency independent of the theatres; theatre owners in Austin compile the databases S and T. Note that T is relatively stable.

We would like to know the answer to: What are the addresses of theatres where Will Smith is playing on Sat. day at 9PM. We write a relational expression for this query and then transform it in several stages to a form which can be efficiciently evaluated. Let predicates

p be $Actor =$ Will Smith
q be $Time =$ Sat. day, 9PM

The query has the form $\pi_{Address}(\sigma_{p \wedge q}(x))$, where x is a relation yet to be defined. Since x has to include information about $Actor$, $Time$ and $Address$, we take x to be $R \bowtie S \bowtie T$. Relation x includes many more attributes than the ones we desire; we will project away the unneeded attributes. The selection operation extracts the tuples which satisfy the predicate $p \wedge q$, and then the projection operation simply lists the addresses. So, the entire query is

$$\pi_{Address}(\sigma_{p \wedge q}\langle R \bowtie S \bowtie T\rangle)$$

Above and in the following expressions, we use brackets of different shapes to help readibility.

We transform this relational expression.

$$
\begin{array}{ll}
 & \pi_{Address}(\sigma_{p \wedge q}\langle R \bowtie S \bowtie T\rangle) \\
\equiv & \{\text{Associativity of join; note that the required conditions are met}\} \\
 & \pi_{Address}(\sigma_{p \wedge q}\langle (R \bowtie S) \bowtie T\rangle) \\
\equiv & \{\text{Selection pushing over join}\} \\
 & \pi_{Address}(\sigma_{p \wedge q}\langle R \bowtie S\rangle \bowtie T) \\
\equiv & \{\text{See lemma below. } p \text{ names only the attributes of } R \text{ and } q \text{ of } S\} \\
 & \pi_{Address}(\langle \sigma_p(R) \bowtie \sigma_q(S)\rangle \bowtie T) \\
\equiv & \{\text{Distributivity of projection over join; } d = \{Theatre\}\} \\
 & \pi_{Address}(\pi_{Theatre}\langle \sigma_p(R) \bowtie \sigma_q(S)\rangle \bowtie \pi_{Address, Theatre}(T)) \\
\equiv & \{\pi_{Address, Theatre}(T) = T\} \\
 & \pi_{Address}(\pi_{Theatre}\langle \sigma_p(R) \bowtie \sigma_q(S)\rangle \bowtie T) \\
\equiv & \{\text{Distributivity of projection over join;} \\
 & \quad \text{the common attribute of } \sigma_p(R) \text{ and } \sigma_q(S) \text{ is } Title\} \\
 & \pi_{Address}(\langle \pi_{Title}(\sigma_p(R)) \bowtie \pi_{Theatre, Title}(\sigma_q(S))\rangle \bowtie T)
\end{array}
$$

Lemma Suppose predicate p names only the attributes of R and q of S. Then,

$$\sigma_{p \wedge q}(R \bowtie S) \;=\; \sigma_p(R) \bowtie \sigma_q(S)$$

Proof:

$$
\begin{array}{ll}
 & \sigma_{p \wedge q}(R \bowtie S) \\
\equiv & \{\text{Selection splitting}\} \\
 & \sigma_p\langle \sigma_q(R \bowtie S)\rangle \\
\equiv & \{\text{Commutativity of join}\} \\
 & \sigma_p\langle \sigma_q(S \bowtie R)\rangle \\
\equiv & \{\text{Selection pushing over join}\} \\
 & \sigma_p\langle \sigma_q(S) \bowtie R\rangle \\
\equiv & \{\text{Commutativity of join}\} \\
 & \sigma_p\langle R \bowtie \sigma_q(S)\rangle \\
\equiv & \{\text{Selection pushing over join}\} \\
 & \sigma_p(R) \bowtie \sigma_q(S)
\end{array}
$$

Compare the original query $\pi_{Address}(\sigma_{p \wedge q}\langle R \bowtie S \bowtie T\rangle)$ with the transformed query $\pi_{Address}(\langle \pi_{Title}(\sigma_p(R)) \bowtie \pi_{Theatre, Title}(\sigma_q(S))\rangle \bowtie T)$ in terms of the efficiency of evaluation. The original query would first compute $R \bowtie S \bowtie T$, a very expensive operation involving three tables. Then selection operation will go over all the tuples again, and the projection incurs a small cost. In the transformed expression, selections are applied as soon as possible, in $\sigma_p(R)$ and $\sigma_q(S)$. This results in much smaller relations, 3 tuples in $\sigma_p(R)$ and 3 in $\sigma_q(S)$. Next, projections will reduce the number of columns in both relations, though not the number of rows. The join of the resulting relation is much more efficient, being applied over smaller tables. Finally, the join with T and projection over *Address* is, again, over smaller tables.

Student Id	Dept	Q1	Q2	Q3
216285932	CS	61	72	49
228544932	CS	35	47	56
859454261	CS	72	68	75
378246719	EE	70	30	69
719644435	EE	60	70	75
549876321	Bus	56	60	52

Table 6.11: Relation *Grades*

6.3.4 Additional Operations on Relations

The operations on relations that have appeared so far are meant to move the data around from one relation to another. There is no way to compute with the data. For example, we cannot ask: How many movies has Will Smith acted in since 1996. To answer such questions we have to count (or add), and none of the operations allow that. We describe two classes of operations, *Aggregation* and *Grouping*, to do such processing. Aggregation operations combine the values in a column in a variety of ways. Grouping creates a number of subrelations from a relation based on some specified attribute values, applies a specified aggregation operation on each, and stores the result in a relation.

Aggregation The following aggregation functions are standard; all except *Count* apply to numbers. For attribute t of a relation,

Count: the number of distinct values (in t)
Sum: sum
Avg: average
Min: minimum
Max: maximum

We write $\mathbf{A}_{f\ t,\ g\ u,\ h\ v\ldots}(R)$ where f, g and h are aggregation functions (shown above) and t, u and v are attribute names in R. The result is a relation which has just one tuple, with values obtained by applying f, g and h to the values of attributes t, u and v of R, respectively. The number of columns in the result is the number of attributes chosen.

Example of Aggregation Consider the *Grades* relation in Table 6.11 (page 157).

Now $\mathbf{A}_{Avg\ Q1}(Grades)$ creates Table 6.12 (page 157).

Avg Q1
59

Table 6.12: Relation *Grades*, Table 6.11, averaged on Q1

Min Q1	Min Q2	Min Q3
35	47	49

Table 6.13: Min of each quiz from relation *Grades*, Table 6.11

We create Table 6.13 (page 158) by $\mathbf{A}_{Min\ Q1,\ Min\ Q2,\ Min\ Q3}(Grades)$. We discuss the names of the attributes of the resulting table, next. □

Consider the names of the attributes in the result Table 6.13, created by $\mathbf{A}_{Min\ Q1,\ Min\ Q2,\ Min\ Q3}(Grades)$. We have simply concatenated the name of the aggregation function and the attribute in forming those names. In general, the user specifies what names to assign to each resulting attribute; we do not develop the notation for such specification here.

Grouping A grouping operation has the form ${}_g\mathbf{A}_L(R)$ where g is a group (see below) and $\mathbf{A}_L(R)$ is the aggregation (L is a list of function, attribute pairs and R is a relation). Whereas $\mathbf{A}_L(R)$ creates a single tuple, ${}_g\mathbf{A}_L(R)$ typically creates multiple tuples. The parameter g is a set of attributes of R. First, R is divided into subrelations R_0, $R_1 \cdots$, based on the attributes g; tuples in each R_i have the same values for g and tuples from different R_is have different values. Then aggregation is applied to each subrelation R_i. The resulting relation has one tuple for each R_i.

Example of Grades, contd. Compute the average score in each quiz for each department. We write ${}_{Dept}\mathbf{A}_{Avg\ Q1,\ Avg\ Q2,\ Avg\ Q3}(Grades)$ to get Table 6.14 (page 158).

Dept	Avg Q1	Avg Q2	Avg Q3
CS	56	62	60
EE	71	49	72
Bus	56	60	52

Table 6.14: Avg of each quiz by department from relation *Grades*, Table 6.11

Count the number of students in each department whose total score exceeds 170: ${}_{Dept}\mathbf{A}_{Count,\ Student\ Id}\langle\pi_{Q1+Q2+Q3>170}(Grades)\rangle$.

6.4 Query Language SQL

A standard in the database community, SQL is a widely used language for data definition and manipulation. SQL statements can appear as part of a C++ program, and, also they can be executed from a command line. A popular version is marketed as MySQL. This section does not give a tutorial on SQL;

in particular, I won't explain the language syntax. I will give a few examples of SQL statements and relate them to relational algebra.

Query facility of SQL is based on relational algebra (most SQL queries can be expressed as relational expressions). But, SQL also provides facilities to insert, delete and update items in a database.

Chapter 7

String Matching

7.1 Introduction

In this chapter, we study a number of algorithms on strings, principally, string matching algorithms. The problem of string matching is to locate all (or some) occurrences of a given *pattern* string within a given *text* string. There are many variations of this basic problem. The pattern may be a set of strings, and the matching algorithm has to locate the occurrence of any pattern in the text. The pattern may be a regular expression for which the "best" match has to be found. The text may consist of a set of strings if, for instance, you are trying to find the occurrence of "to be or not to be" in the works of Shakespeare. In some situations the text string is fixed, but the pattern changes, as in searching Shakespeare's works. Quite often, the goal is not to find an exact match but a close enough match, as in DNA sequences or Google searches.

The string matching problem is quite different from dictionary or database search. In dictionary search, you are asked to determine if a given word belongs to a set of words. Usually, the set of words —the dictionary— is fixed. A hashing algorithm suffices in most cases for such problems. Database searches can be more complex than exact matches over strings. The database entries may be images (say, thumbprints), distances among cities, positions of vehicles in a fleet or salaries of individuals. A query may involve satisfying a predicate, e.g., find any hospital that is within 10 miles of a specific vehicle and determine the shortest path to it.

We spend most of this chapter on the exact string matching problem: given a text string t and a pattern string p over some alphabet, construct a list of positions where p occurs within t. See Table 7.1 for an example.

The naive algorithm for this problem matches the pattern against the string starting at every possible position in the text. This may take $O(m \times n)$ time where m and n are the two string lengths. We show three different algorithms all of which run much faster, and one is an $O(m + n)$ algorithm.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
text	a	g	c	t	t	a	c	g	a	a	c	g	t	a	a	c	g	a
pattern	a	a	c	g														
output									*					*				

Table 7.1: The pattern matching problem

Exercise 68

You are given strings x and y of equal length, and asked to determine if x is a rotation of y. Solve the problem through string matching. □

Solution Determine if x occurs as a substring in yy.

Notation Let the text be t and the pattern be p. The symbols in a string are indexed starting at 0. We write $t[i]$ for the ith symbol of t, and $t[i..j]$ for the substring of t starting at i and ending just before j, $i \leq j$. Therefore, the length of $t[i..j]$ is $j - i$; it is empty if $i = j$. Similar conventions apply to p.

For a string r, write $\overline{r}$ for its length. Henceforth, the length of the pattern p, $\overline{p}$, is m; so, its elements are indexed 0 through $m - 1$. Text t is an infinite string. This assumption is made to avoid worrying about the termination of the algorithm. We simply show that every substring in t that matches p will be found ultimately.

7.2 Rabin-Karp Algorithm

The idea of this algorithm is based on hashing. Given text t and pattern p, compute $val(p)$, where function val will be specified later. Then, for each substring s of t whose length is $\overline{p}$, compute $val(s)$. Since

$$p = s \;\Rightarrow\; val(p) = val(s), \text{ or}$$
$$val(p) \neq val(s) \;\Rightarrow\; p \neq s$$

we may discard string s if $val(p) \neq val(s)$. We illustrate the procedure for strings of 5 decimal digits where val returns the sum of the digits in its argument string.

Let $p = 27681$; so, $val(p) = 24$. Consider the text given in Table 7.2; the function values are also shown there (at the position at which a string ends). There are two strings for which the function value is 24, namely 27681 and 19833. We compare each of these strings against the original string, 27681, to find that there is one exact match.

Function val is similar to a hash function. It is used to remove most strings from consideration. Only when $val(s) = val(p)$ do we have a *collision*, and we match s against p. As in hashing, we require that there should be very few collisions on the average; moreover, val should be easily computable incrementally, i.e., from one string to the next.

text	2	4	1	5	7	2	7	6	8	1	9	8	3	3	7	8	1	4
$val(s)$					19	19	22	27	30	24	31	32	29	24	30	29	22	23

Table 7.2: Computing function values in Rabin-Karp algorithm

Minimizing Collisions A function like *val* partitions the set of strings into equivalence classes: two strings are in the same equivalence class if their function values are identical. Strings in the same equivalence class cause collisions, as in the case of 27681 and 19833, shown above. In order to reduce collisions, we strive to make the equivalence classes equal in size. Then, the probability of collision is $1/n$, where n is the number of possible values of *val*.

For the function that sums the digits of a 5-digit number, the possible values range from 0 (all digits are 0s) to 45 (all digits are 9s). But the 46 equivalence classes are not equal in size. Note that $val(s) = 0$ iff $s = 00000$; thus if you are searching for pattern 00000 you will never have a collision. However, $val(s) = 24$ for 5875 different 5-digit strings. So, the probability of collision is around 0.05875 (since there are 10^5 5-digit strings). If there had been an even distribution among the 46 equivalence classes, the probability of collision would have been 1/46, or around 0.02173, almost three times fewer collisions than when $val(s) = 24$.

One way of distributing the numbers evenly is to let $val(s) = s \bmod q$, for some prime q. This function distributes the numbers evenly among q bins, 0 through $q - 1$. Since the number of 5-digit strings may not be a multiple of q, not all equivalence classes will be of the same size, but no two classes differ by more than 1 in their sizes. So, this is as good as it gets.

Incremental computation of *val* The next question is how to calculate *val* efficiently, for *all* substrings in the text. If the function adds up the digits in 5-digit strings, then it is easy to compute: suppose we have already computed the sum, s, for a five digit string $b_0b_1b_2b_3b_4$; to compute the sum for the next substring $b_1b_2b_3b_4b_5$, we assign $s := s - b_0 + b_5$. I show that the modulo function can be calculated equally easily.

The main observation for performing this computation is as follows. Suppose we have already scanned a n-digit string "ar", where a is the first symbol of the string and r is its tail; let ar denote the numerical value of "ar". The function value, $ar \bmod q$, has been computed already. When we scan the digit b following r, we have to evaluate $rb \bmod q$ where rb is the numerical value of "rb". We represent rb in terms of ar, a and b. First, remove a from ar by subtracting $a \times 10^{n-1}$; this gives us r. Next, left shift r by one position, which is $r \times 10$. Finally, add b. So, $rb = (ar - a \times 10^{n-1}) \times 10 + b$. To compute $rb \bmod q$, for prime q, we need a few simple results about mod.

$$
\begin{aligned}
(a + b) \bmod q &= (a + b \bmod q) \bmod q \\
(a - b) \bmod q &= (a - b \bmod q) \bmod q \\
(a \times b) \bmod q &= (a \times b \bmod q) \bmod q
\end{aligned}
$$

Modular Simplification Rule

Let e be any expression over integers that has only addition, subtraction, multiplication and exponention as its operators. Let e' be obtained from e by replacing any subexpression t of e by $(t \bmod p)$. Then, $e \overset{\text{mod } p}{\equiv} e'$, i.e., $e \bmod p = e' \bmod p$.

Note that an exponent is not a subexpression; so, it can't be replaced by its mod.

Examples

$(20 + 5) \bmod 3 = ((20 \bmod 3) + 5) \bmod 3$
$((x \times y) + g) \bmod p = (((x \bmod p) \times y) + (g \bmod p)) \bmod p$
$x^n \bmod p = (x \bmod p)^n \bmod p$
$x^{2n} \bmod p = (x^2)^n \bmod p = (x^2 \bmod p)^n \bmod p$
$x^n \bmod p = x^{n \bmod p} \bmod p$, is wrong. □

We use this rule to compute $rb \bmod q$.

$$
\begin{array}{ll}
 & rb \bmod q \\
= & \{rb = (ar - a \times 10^{n-1}) \times 10 + b\} \\
 & ((ar - a \times 10^{n-1}) \times 10 + b) \bmod q \\
= & \{\text{replace } ar \text{ and } 10^{n-1}\} \\
 & (((ar \bmod q) - a \times (10^{n-1} \bmod q)) \times 10 + b) \bmod q \\
= & \{\text{let } u = ar \bmod q \text{ and } f = 10^{n-1} \bmod q, \text{ both already computed}\} \\
 & ((u - a \times f) \times 10 + b) \bmod q
\end{array}
$$

Example Let q be 47. Suppose we have computed $12768 \bmod 47$, which is 31. And, also $10^4 \bmod 47$ which is 36. We compute $27687 \bmod 47$ by

$$
\begin{array}{ll}
 & ((31 - 1 \times 36) \times 10 + 7) \bmod 47 \\
= & ((-5) \times 10 + 7) \bmod 47 \\
= & (-43) \bmod 47 \\
= & 4
\end{array}
$$
□

Exercise 69

Show that the equivalence classes under mod q are almost equal in size. □

Exercise 70

Derive a general formula for incremental calculation when the alphabet has d symbols, so that each string can be regarded as a d-ary number. □

7.3 Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt algorithm (KMP) locates all occurrences of a pattern in a text in linear time (in the combined lengths of the two strings). It is a refined version of the naive algorithm.

7.3.1 Informal Description

Let the pattern be "JayadevMisra". Suppose, we have matched the portion "JayadevM" against some part of the text string, but the next symbol in the text differs from 'i', the next symbol in the pattern. The naive algorithm would shift one position beyond 'J' in the text, and start the match all over, starting with 'J', the first symbol of the pattern. The KMP algorithm is based on the observation that no symbol in the text that we have already matched with "JayadevM" can possibly be the start of a full match: we have just discovered that the there is no match starting at 'J', and there is no match starting at any other symbol because none of them is a 'J'. So, we may skip this entire string in the text and shift to the next symbol beyond "JayadevM" to begin a match.

In general, we will not be lucky enough to skip the entire piece of text that we had already matched, as we could in the case of "JayadevM". For instance, suppose the pattern is "axbcyaxbts", and we have already matched "axbcyaxb"; see Table 7.3. Suppose the next symbol in the text does not match 't', the next symbol in the pattern. A possible match could begin at the second occurrence of 'a', because the text there is "axb", a prefix of the pattern. So, we shift to that position in the text, but we avoid scanning any symbol in this portion of the text again. The formal description, given next, establishes the conditions that need to be satisfied for this scheme to work.

7.3.2 Algorithm Outline

At any point during the algorithm we have matched a portion of the pattern against the text; that is, we maintain the following invariant where l and r are indices in t defining the two ends of the matched portion.

KMP-INV:
$l \leq r \ \wedge \ t[l..r] = p[0..r-l]$, and
all occurrences of p starting prior to l in the text have been located.

The invariant is established initially by setting

$l, r := 0, 0$

In subsequent steps we compare the next symbols from the text and the pattern. If there is no next symbol in the pattern, we have found a match, and we discuss what to do next below. For the moment, assume that p has a next symbol, $p[r-l]$.

$t[r] = p[r-l] \quad \longrightarrow \ r := r+1$
{ more text has been matched }
$t[r] \neq p[r-l] \ \wedge \ r = l \quad \longrightarrow \ l := l+1; \ r := r+1$
{ we have an empty string matched so far;
the first pattern symbol differs from the next text symbol }
$t[r] \neq p[r-l] \ \wedge \ r > l \quad \longrightarrow \ l := l'$
{ a nonempty prefix of p has matched but the next symbols don't }

Note that in the third case, r is not changed; so, none of the symbols in $t[l'..r]$ will be scanned again.

The question (in the third case) is, what is l'? Abbreviate $t[l..r]$ by v and $t[l'..r]$ by u. We show below that u is a proper prefix *and* a proper suffix of v. Thus, l' is given by the longest u that is both a proper prefix and a proper suffix of v.

From the invariant, v is a prefix of p. Also, from the invariant, u is a prefix of p, and, since $l' > l$, u is a shorter prefix than v. Therefore, u is a proper prefix of v. Next, since their right ends match, $t[l'..r]$ is a proper suffix of $t[l..r]$, i.e., u is a proper suffix of v.

We describe the algorithm in schematic form in Table 7.3. Here, we have already matched the prefix "axbcyaxb", which is v. There is a mismatch in the next symbol. We shift the pattern so that the prefix "axb", which is u, is aligned with a portion of the text that matches it.

index	l					l'			r						
text	a	x	b	c	y	a	x	b	z	-	-	-	-	-	-
pattern	a	x	b	c	y	a	x	b	t	s					
newmatch						a	x	b	c	y	a	x	b	t	s

Table 7.3: Matching in the KMP algorithm

In general, there may be many strings, u, which are both proper prefix and suffix of v; in particular, the empty string satisfies this condition for any v. Which u should we choose? Any u we choose could possibly lead to a match, because we have not scanned beyond $t[r]$. So, we increment l by the minimum required amount, i.e., u is the longest string that is both a proper prefix and suffix of v; we call u the *core* of v.

The question of computing l' then reduces to the following problem: given a string v, find its core. Then $l' = l + (\text{length of } v) - (\text{length of core of } v)$. Since v is a prefix of p, we precompute the cores of all prefixes of the pattern, so that we may compute l' whenever there is a failure in the match. In the next subsection we develop a linear algorithm to compute cores of all prefixes of the pattern.

After the pattern has been completely matched, we record this fact and let $l' = l + (\text{length of } p) - (\text{length of core of } p)$.

We show that KMP runs in linear time. Observe that $l + r$ increases in each step (in the last case, $l' > l$). Both l and r are bounded by the length of the text string; so the number of steps is bounded by a linear function of the length of text. The core computation, in the next section, is linear in the size of the pattern. So, the whole algorithm is linear.

7.3.3 The Theory of Core

First, we develop a small theory about prefixes and suffixes from which the computation of the core will follow.

For strings u and v, we write $u \preceq v$ to mean that u is both a prefix and a suffix of v. Observe that $u \preceq v$ holds whenever u is a prefix of v and the reverse of u is a prefix of the reverse of v. The following properties of $\preceq$ follow from the properties of the prefix relation; you are expected to develop the proofs. Henceforth, u and v denote arbitrary strings and ϵ is the empty string.

1. $\epsilon \preceq u$.
2. $\preceq$ is a partial order.

As is usual, we write $u \prec v$ to mean that $u \preceq v$ and $u \neq v$; in that case we say u is *below* v.

Exercise 71
Find all strings below *ababab*. □

Exercise 72

1. Show that $\preceq$ is a partial order. Use the fact that prefix relation is a partial order.
2. Show that for any v there is a total order among all u where $u \preceq v$, i.e.,

$$(u \preceq v \ \wedge \ w \preceq v) \ \Rightarrow \ (u \preceq w \ \vee \ w \preceq u)$$ □

Definition of core For any nonempty v, *core* of v, written as $c(v)$, is the longest string below v. The core is defined for every v, $v \neq \epsilon$, because there is at least one string, namely ϵ, that is a proper prefix and suffix of every nonempty string.

Example We compute the cores of several strings.

String	Core
a	ϵ
ab	ϵ
abb	ϵ
abba	a
abbab	ab
abbabb	abb
abbabba	abba
abbabbb	ϵ

□

The traditional way to define core is as follows: (1) $c(v) \prec v$, and (2) for any w where $w \prec v$, $w \preceq c(v)$. We give a different, though equivalent, definition that is more convenient for formal manipulations. For any u and v, $v \neq \epsilon$,

(core definition): $u \preceq c(v) \ \equiv \ u \prec v$

It follows, by replacing u with $c(v)$, that $c(v) \prec v$. In particular, $\overline{c(v)} < \overline{v}$.

Exercise 73
Show that every non-empty string has a unique core.

Solution Let r and s be cores of v. We show that $r = s$. For any u

$$\begin{array}{ll} & u \preceq r \\ \equiv & \{r \text{ is a core; use definition of core}\} \\ & u \prec v \\ \equiv & \{s \text{ is a core; use definition of core}\} \\ & u \preceq s \end{array}$$

From above, $u \preceq r \ \equiv \ u \preceq s$, for all u. Setting u to r, we get $r \preceq r \ \equiv \ r \preceq s$, i.e., $r \preceq s$. Similarly, we can deduce $s \preceq r$. So, $r = s$ from the antisymmetry of $\preceq$. □

Exercise 74

Compute cores of all prefixes of *ababab*. □

Exercise 75

Let u be a longer string than v. Is $c(u)$ necessarily longer than $c(v)$? □

We write $c^i(v)$ for i-fold application of c to v, i.e., $c^i(v) = \overbrace{c(c(..(c}^{i \text{ times}}(v)..)))$ and $c^0(v) = v$. Since $\overline{c(v)} < \overline{v}$, $c^i(v)$ is defined only for some i, not necessarily all i, in the range $0 \leq i \leq \overline{v}$. Note that, $c^{i+1}(v) \prec c^i(v) \ldots c^1(v) \prec c^0(v) = v$.

Exercise 76

Compute $c^i(ababab)$ for all possible i. □

The following proposition says that any string below v can be obtained by applying the function c a sufficient number of times to v.

P1: For any u and v,

$$u \preceq v \ \equiv \ \langle \exists i : 0 \leq i : u = c^i(v) \rangle$$

Proof: The proof is by induction on the length of v.

• $\overline{v} = 0$:

$$\begin{array}{ll} & u \preceq v \\ \equiv & \{\overline{v} = 0, \text{ i.e., } v = \epsilon\} \\ & u = \epsilon \ \wedge \ v = \epsilon \\ \equiv & \{\text{definition of } c^0\text{: } v = \epsilon \ \Rightarrow \ c^i(v) \text{ is defined for } i = 0 \text{ only}\} \\ & \langle \exists i : 0 \leq i : u = c^i(v) \rangle \end{array}$$

• $\overline{v} = n + 1$, $n \geq 0$:

$$
\begin{array}{ll}
 & u \preceq v \\
\equiv & \{\text{definition of } \preceq\} \\
 & u = v \ \vee \ u \prec v \\
\equiv & \{\text{definition of core}\} \\
 & u = v \ \vee \ u \preceq c(v) \\
\equiv & \{\overline{c(v)} < \overline{v}\text{; induction hypothesis on second term}\} \\
 & u = v \ \vee \ \langle \exists i : 0 \leq i : u = c^i(c(v)) \rangle \\
\equiv & \{\text{rewrite}\} \\
 & u = c^0(v) \ \vee \ \langle \exists i : 0 < i : u = c^i(v) \rangle \\
\equiv & \{\text{rewrite}\} \\
 & \langle \exists i : 0 \leq i : u = c^i(v) \rangle
\end{array}
$$

□

Corollary For any u and v, $v \neq \epsilon$,

$$u \prec v \ \equiv \ \langle \exists i : 0 < i : u = c^i(v) \rangle$$

□

Exercise 77

Show that the core function is monotonic, that is,

$$u \preceq v \ \Rightarrow \ c(u) \preceq c(v)$$

□

An abstract program for the core computation

We develop a program to compute the cores of all nonempty prefixes of a given string p. First, we present an abstract version, and, later, a concrete version.

Notation Henceforth, p is a non-empty string, u and v are proper prefixes of p, and u' and v' their one-symbol extensions. So, u' and v' are prefixes of p. We write ϵ' for the string containing the first symbol of p. Recall that $p[\overline{u}]$ is the last symbol of u' and $p[\overline{v}]$ of v'. □

The structure of the program is:

$v := \epsilon'$; $c(v) := \epsilon$;
while v is a proper prefix of p **do**
  {cores of all prefixes of p up to and including v have been computed}
  compute $c(v')$;
  $v := v'$
endwhile

The following lemma is the basis for computing $c(v')$.

Lemma $u' \prec v' \ \equiv \ u \prec v \ \wedge \ p[\overline{u}] = p[\overline{v}]$
Proof:

$$
\begin{array}{ll}
 & u' \prec v' \\
\equiv & \{\text{definition of } \prec\} \\
 & u' \text{ is a proper prefix of } v',\ u' \text{ is a proper suffix of } v' \\
\equiv & \{\text{given that } u' \text{ and } v' \text{ are prefixes of } p, \\
 & \quad u' \text{ is a proper prefix of } v' \ \equiv\ u \text{ is a proper prefix of } v\} \\
 & u \text{ is a proper prefix of } v,\ u' \text{ is a proper suffix of } v' \\
\equiv & \{\text{definition of suffix}\} \\
 & u \text{ is a proper prefix of } v,\ u \text{ is a proper suffix of } v,\ p[\overline{u}] = p[\overline{v}] \\
\equiv & \{\text{definition of } \prec\} \\
 & u \prec v \ \wedge\ p[\overline{u}] = p[\overline{v}] \qquad \square
\end{array}
$$

This lemma tells us that every nonempty string u' that is below v' satisfies

$$u \prec v \ \wedge\ p[\overline{u}] = p[\overline{v}] \qquad (*)$$

If no string u satisfies (*), there is no nonempty string below v'; so, $c(v') = \epsilon$. Otherwise, the longest string satisfying (*) defines $c(v')$.

To find the longest string satisfying (*), we enumerate all strings u satisfying $u \prec v$ in decreasing order of length, and for each one check if $p[\overline{u}] = p[\overline{v}]$. The first string for which this test succeeds is the desired u and $c(v') = u'$; if the test fails for all u, $c(v') = \epsilon$. Proposition (P1) tells us how to enumerate strings below v in decreasing order of length: enumerate $c^1(v), c^2(v), \ldots, \epsilon$. This enumeration is easy because cores of all prefixes of p up to and including v have already been computed. The complete program is given below.

```
v := ε'; c(v) := ε;
while v is a proper prefix of p do
  u := c(v);
  while p[ū] ≠ p[v̄] ∧ u ≠ ε do
    {u = c^i(v), for some i, i > 0}
    u := c(u)
  endwhile ;
  {u = c^i(v), for some i, i > 0 and (p[ū] = p[v̄] ∨ u = ε)}
  if p[ū] = p[v̄]
    then c(v') := u'
    else c(v') := ε
  endif ;
  v := v'
endwhile
```

A concrete program for the core computation

We systematically transform the abstract program given above to obtain a concrete program that can be directly implemented. Since u and v are prefixes of p, we represent them using indices into p, i and j, respectively. That is,

$$
\begin{array}{ll}
u = p[0..i] & \text{and } v = p[0..j], \text{ so, since } p[0..i] \text{ does not include } p[i], \\
\overline{u} = i & \text{and } \overline{v} = j, \text{ and} \\
\overline{u'} = i + 1 & \text{and } \overline{v'} = j + 1.
\end{array}
$$

We store $c(v)$, for all v, $v \neq \epsilon$, in an array d, where $d[k]$ is the length of the core of $p[0..k]$. Therefore,

$d[j] = \overline{c(v)}$, since $v = p[0..j]$, and
$d[i] = \overline{c(u)}$ and $d[j+1] = \overline{c(v')}$

The resulting program is given below.

```
j := 1; d[1] := 0;
while j < p̄ do
  i := d[j];
  while p[i] ≠ p[j] ∧ i ≠ 0 do
    i := d[i]
  endwhile ;
  { p[i] = p[j] ∨ i = 0 }
  if p[i] = p[j]
    then d[j+1] := i+1
    else d[j+1] := 0
  endif ;
  j := j+1
endwhile
```

Analysis of the running time of the core computation

We show that the program of Section 7.3.3 runs in linear time in the length of the pattern p. We transform the program to a loop consisting of three guarded commands, as shown below, which is more easily analyzed. We have labeled the guarded commands S_1, S_2 and S_3, for easy reference.

```
j := 1; d[1] := 0; i := d[j];
loop
  S1:: j < p̄ ∧ p[i] ≠ p[j] ∧ i ≠ 0  →  i := d[i]
  S2:: j < p̄ ∧ p[i] = p[j]          →  d[j+1] := i+1; j := j+1; i := d[j]
  S3:: j < p̄ ∧ p[i] ≠ p[j] ∧ i = 0  →  d[j+1] := 0; j := j+1; i := d[j]
endloop
```

We show below that execution of each guarded command increases $2j - i$. Since $j \leq \overline{p}$ and $i \geq 0$ (prove these as invariants), $2j - i$ never exceeds $2\overline{p}$. Initially, $2j - i = 2$. Therefore, the number of executions of all guarded commands is $O(\overline{p})$.

Now, we consider the right side of each guarded command S_k, $1 \leq k \leq 3$, and show that each one strictly increases $2j - i$, i.e.,

$\{2j - i = n\}$ right side of S_k $\{2j - i > n\}$

• Proof for S_1:

$\{2j - i = n\}\ i := d[i]\ \{2j - i > n\}$
, goal
$2j - i = n\ \Rightarrow\ 2j - d[i] > n$, axiom of assignment
$2j - i < 2j - d[i]$, arithmetic
$d[i] < i$, arithmetic
true , $i = \overline{u}$ and $d[i] = \overline{c(u)}$ □

• Proof for S_2:

$\{2j - i = n\}\ d[j+1] := i + 1;\ j := j + 1;\ i := d[j]\ \{2j - i > n\}$
, goal
$\{2j - i = n\}\ d[j+1] := i + 1;\ j := j + 1\ \{2j - d[j] > n\}$
, axiom of assignment
$\{2j - i = n\}\ d[j+1] := i + 1\ \{2(j+1) - d[j+1] > n\}$
, axiom of assignment
$\{2j - i = n\}\ \{2(j+1) - (i+1) > n\}$
, axiom of assignment
$2j - i = n\ \Rightarrow\ 2(j+1) - (i+1) > n$
, simplify
true , arithmetic □

• Proof for S_3:

$\{2j - i = n\}\ d[j+1] := 0;\ j := j + 1;\ i := d[j]\ \{2j - i > n\}$
, goal
$\{2j - i = n\}\ d[j+1] := 0;\ j := j + 1\ \{2j - d[j] > n\}$
, axiom of assignment
$\{2j - i = n\}\ d[j+1] := 0\ \{2(j+1) - d[j+1] > n\}$
, axiom of assignment
$\{2j - i = n\}\ \{2(j+1) - 0 > n\}$
, axiom of assignment
$2j - i = n\ \Rightarrow\ 2(j+1) > n$, simplify
true , arithmetic and $i \geq 0$ (invariant) □

Exercise 78

1. Show that you can match pattern p against text t by computing the cores of all prefixes of pt (pt is the concatenation of p and t).

2. Define u to be the k-core of string v, $k \geq 0$, $v \neq \epsilon$, if u is the longest string below v whose length is at most k. Show that the k-core is well-defined. Devise an algorithm to compute the k-core of a string for a given k. □

7.4 Boyer-Moore Algorithm

The next string matching algorithm we study is due to Boyer and Moore. It has the best performance, on the average, of all known algorithms for this problem. In many cases, it runs in sublinear time, because it may not even scan all the symbols of the text. Its worst case behavior could be as bad as the naive matching algorithm.

At any moment, imagine that the pattern is *aligned* with a portion of the text of the same length, though only a part of the aligned text may have been matched with the pattern. Henceforth, alignment refers to the substring of t that is aligned with p and l is the index of the left end of the alignment; i.e., $p[0]$ is aligned with $t[l]$ and, in general, $p[i]$, $0 \leq i < m$, with $t[l+i]$. Whenever there is a mismatch, the pattern is *shifted* to the right, i.e., l is increased, as explained in the following sections.

7.4.1 Algorithm Outline

The overall structure of the program is a loop that has the invariant:

Q1: Every occurrence of p in t that starts before l has been recorded.

The following loop records every occurrence of p in t eventually.

$l := 0$;
{ Q1 }
loop
  { Q1 }
  "increase l while preserving Q1"
  { Q1 }
endloop

Next, we show how to increase l while preserving Q1. To do so, we need to match certain symbols of the pattern against the text. We introduce variable j, $0 \leq j < m$, with the meaning that the suffix of p starting at position j matches the corresponding portion of the alignment; i.e.,

Q2: $0 \leq j \leq m$, $p[j..m] = t[l+j..l+m]$

Thus, the whole pattern is matched when $j = 0$, and no part has been matched when $j = m$.

We establish Q2 by setting j to m. Then, we match the symbols from *right to left* of the pattern (against the corresponding symbols in the alignment) until we find a mismatch or the whole pattern is matched.

$j := m$;
{ Q2 }
while $j > 0 \ \wedge \ p[j-1] = t[l+j-1]$ **do** $j := j-1$ **endwhile**

```
{ Q1 ∧ Q2 ∧ (j = 0 ∨ p[j − 1] ≠ t[l + j − 1]) }
if j = 0
   then { Q1 ∧ Q2 ∧ j = 0 } record a match at l;      l := l′ { Q1 }
   else { Q1 ∧ Q2 ∧ j > 0 ∧ p[j − 1] ≠ t[l + j − 1] } l := l″{ Q1 }
endif
{ Q1 }
```

Next, we show how to compute l and l', $l' > l$ and $l'' > l$, so that Q1 is satisfied. For better performance, l should be increased as much as possible in each case. We take up the computation of l'' next; computation of l' is a special case of this.

The precondition for the computation of l'' is,

$$\text{Q1} \wedge \text{Q2} \wedge j > 0 \ \wedge \ p[j-1] \neq t[l+j-1].$$

We consider two heuristics, each of which can be used to calculate a value of l''; the greater value is assigned to l. The first heuristic, called the *bad symbol heuristic*, exploits the fact that we have a mismatch at position $j-1$ of the pattern. The second heuristic, called the *good suffix heuristic*, uses the fact that we have matched a suffix of p with the suffix of the alignment, i.e., $p[j..m] = t[l+j..l+m]$ (though the suffix may be empty).

7.4.2 The Bad Symbol Heuristic

Suppose we have the pattern "attendance" that we have aligned against a portion of the text whose suffix is "hce", as shown in Table 7.4.

text	-	-	-	-	-	-	-	h	c	e										
pattern	a	t	t	e	n	d	a	n	c	e										
align									a	t	t	e	n	d	a	n	c	e		

Table 7.4: The bad symbol heuristic

The suffix "ce" has been matched; the symbols 'h' and 'n' do not match. We now reason as follows. If symbol 'h' of the text is part of a full match, that symbol has to be aligned with an 'h' of the pattern. There is no 'h' in the pattern; so, no match can include this 'h' of the text. Hence, the pattern may be shifted to the symbol following 'h' in the text, as shown under *align* in Table 7.4. Since the index of 'h' in the text is $l+j-1$ (that is where the mismatch occurred), we have to align $p[0]$ with $t[l+j]$, i.e., l should be increased to $l+j$. Observe that we have shifted the alignment several positions to the right without scanning the text symbols shown by dashes, '-', in the text; this is how the algorithm achieves sublinear running time in many cases.

Next, suppose the mismatched symbol in the text is 't', as shown in Table 7.5.

text	-	-	-	-	-	-	-	t	c	e
pattern	a	t	t	e	n	d	a	n	c	e

Table 7.5: The bad symbol heuristic

Unlike 'h', symbol 't' appears in the pattern. We align some occurrence of 't' in the pattern with that in the text. There are two possible alignments, which we show in Table 7.6.

text	-	-	t	c	e	-	-	-	-	-	-
align1	a	t	t	e	n	d	a	n	c	e	
align2		a	t	t	e	n	d	a	n	c	e

Table 7.6: New alignment in the bad symbol heuristic

Which alignment should we choose? The same question also comes up in the good suffix heuristic. We have several possible shifts each of which matches a portion of the alignment. We adopt the following rule for shift:

Minimum shift rule: Shift the pattern by the minimum allowable amount.

According to this rule, in Table 7.7 we would shift the pattern to get *align1*.

Justification for the rule: This rule preserves Q1; we never skip over a possible match following this rule, because no smaller shift yields a match at the given position, and, hence no full match.

Conversely, consider the situation shown in Table 7.7. The first pattern line shows an alignment where there is a mismatch at the rightmost symbol in the alignment. The next two lines show two possible alignments that correct the mismatch. Since the only text symbol we have examined is 'x', each dash in Table 7.7 could be any symbol at all; so, in particular, the text could be such that the pattern matches against the first alignment, *align1*. Then, we will violate invariant Q1 if we shift the pattern as shown in *align2*. □

text	-	-	x	-	-
pattern	x	x	y		
align1		x	x	y	
align2			x	x	y

Table 7.7: Realignment in the good suffix heuristic

For each symbol in the alphabet, we precalculate its rightmost position in the pattern. The rightmost 't' in "attendance" is at position 2. To align the mismatched 't' in the text in Table 7.6 that is at position $t[l+j-1]$, we align $p[2]$

with $t[l+j-1]$, that is, $p[0]$ with $t[l-2+j-1]$. In general, if the mismatched symbol's rightmost occurrence in the pattern is at $p[k]$, then $p[0]$ is aligned with $t[l-k+j-1]$, or l is increased by $-k+j-1$. For a nonexistent symbol in the pattern, like 'h', we set its rightmost occurrence to -1 so that l is increased to $l+j$, as required.

The quantity $-k+j-1$ is negative if $k > j-1$. That is, the rightmost occurrence of the mismatched symbol in the pattern is to the right of the mismatch. Fortunately, the good suffix heuristic, which we discuss in Section 7.4.3, always yields a positive increment for l; so, we ignore this heuristic if it yields a negative increment.

Computing the rightmost positions of the symbols in the pattern For a given alphabet, we compute an array rt, indexed by the symbols of the alphabet, so that for any symbol 'a',

$$rt(\text{'a'}) = \begin{cases} \text{position of the rightmost 'a' in } p, & \text{if 'a' is in } p \\ -1 & \text{otherwise} \end{cases}$$

The following simple loop computes rt.

let rt['a'] := -1, for every symbol 'a' in the alphabet;
for $j = 0$ **to** $m-1$ **do**
  $rt[p[j]] := j$
endfor

7.4.3 The Good Suffix Heuristic

Suppose we have a pattern "abxabyab" of which we have already matched the suffix "ab", but there is a mismatch with the preceding symbol 'y', as shown in Table 7.8.

text	-	-	-	-	-	z	a	b	-	-
pattern	a	b	x	a	b	y	a	b		

Table 7.8: A good suffix heuristic scenario

Then, we shift the pattern to the right so that the matched part is occupied by the same symbols, "ab"; this is possible only if there is another occurrence of "ab" in the pattern. For the pattern of Table 7.8, we can form the new alignment in two possible ways, as shown in Table 7.9.

No complete match of the suffix s is possible if s does not occur elsewhere in p. This possibility is shown in Table 7.10, where s is "xab". In this case, the best that can be done is to match with a suffix of "xab", as shown in Table 7.10. Note that the matching portion "ab" is a prefix of p. Also, it is a suffix of p, being a suffix of "xab", that is a suffix of p.

text	-	-	z	a	b	-	-	-	-	-	-
align1	a	b	x	a	b	y	a	b			
align2				a	b	x	a	b	y	a	b

Table 7.9: Realignment in the good suffix heuristic

text	-	-	x	a	b	-	-	-
pattern	a	b	x	a	b			
align				a	b	x	a	b

Table 7.10: The matched suffix is nowhere else in p

As shown in the preceding examples, in all cases we shift the pattern to align the right end of a proper prefix r with the right end of the previous alignment. Also, r is a suffix of s or s is a suffix of r. In the example in Table 7.9, s is "ab" and there are two possible r, "abxab" and "ab", for which s is a suffix. Additionally, ϵ is a suffix of s. In Table 7.10, s is "xab" and there is exactly one nonempty r, "ab", which is a suffix of s. Let

$$R = \{r \text{ is a proper prefix of } p \wedge (r \text{ is a suffix of } s \vee s \text{ is a suffix of } r)\}$$

The good suffix heuristic aligns an r in R with the end of the previous alignment, i.e., the pattern is shifted to the right by $m - \overline{r}$. Let $b(s)$ be the amount by which the pattern should be shifted for a suffix s. According to the minimum shift rule,

$$b(s) = \min\{m - \overline{r} \mid r \in R\}$$

In the rest of this section, we develop an efficient algorithm for computing $b(s)$.

Shifting the pattern in the algorithm of Section 7.4.1

In the algorithm outlined in Section 7.4.1, we have two assignments to l, the assignment

$l := l'$, when the whole pattern has matched, and
$l := l''$, when $p[j..m] = t[l+j..l+m]$ and $p[j-1] \neq t[l+j-1]$

$l := l'$ is implemented by
  $l := l + b(p)$, and
$l := l''$ is implemented by
  $l := l + \max(b(s), j - 1 - rt(h))$,
    where $s = p[j..m]$ and $h = t[l+j-1]$

Properties of $b(s)$, the shift amount for suffix s

We repeat the definition of $b(s)$.

$$b(s) = \min\{m - \overline{r} \mid r \in R\}$$

Notation We abbreviate $\min\{m - \overline{r} \mid r \in R\}$ to $\min(m - R)$. In general, let S be a set of strings and $e(S)$ an expression that includes S as a term. Then, $\min(e(S)) = \min\{e(\overline{i}) \mid i \in S\}$, where $e(\overline{i})$ is obtained from e by replacing S by $\overline{i}$. □

Rewrite R as $R' \cup R''$, where

$$\begin{aligned} R' &= \{r \text{ is a proper prefix of } p \ \wedge \ r \text{ is a suffix of } s\} \\ R'' &= \{r \text{ is a proper prefix of } p \ \wedge \ s \text{ is a suffix of } r\} \end{aligned}$$

Then,

$$b(s) = \min(\min(m - R'),\ \min(m - R''))$$

where minimum over empty set is ∞.

P1: R is nonempty and $b(s)$ is well-defined.
Proof: Note that $\epsilon \in R'$ and $R = R' \cup R''$. Then, from its definition, $b(s)$ is well-defined. □

P2: $c(p) \in R$
Proof: From the definition of core, $c(p) \prec p$. Hence, $c(p)$ is a proper prefix of p. Also, $c(p)$ is a suffix of p, and, since s is a suffix of p, they are totally ordered. So, either $c(p)$ is a suffix of s or s is a suffix of $c(p)$. Hence, $c(p) \in R$. □

P3: $\min(m - R') \ \geq \ m - \overline{c(p)}$
Proof: Consider any r in R'. Since r is a suffix of s and s is a suffix of p, r is a suffix of p. Also, r is a proper prefix of p. So, $r \prec p$. From the definition of core, $r \preceq c(p)$. Hence, $m - \overline{r} \geq \ m - \overline{c(p)}$ for every r in R'. □

P4: Let $V = \{v \mid v \text{ is a suffix of } \ p \ \wedge \ c(v) = s\}$.
Then, $\min(m - R'') \ = \ \min(V - \overline{s})$
Proof: Note that R'' may be empty. In that case, V will be empty too and both will have the same minimum, ∞. This causes no problem in computing $b(s)$ because, from (P1), R is nonempty.

Consider any r in R''. Note that r is a prefix of p and has s as a suffix; so $p = xsy$, for some x and y, where $r = xs$ and y is the remaining portion of p. Also, $y \neq \epsilon$ because r is a proper prefix. Let u stand for sy; then u is a suffix of p. And, $u \neq \epsilon$ because $y \neq \epsilon$, though u may be equal to p, because x could be empty. Also, s is a prefix of u ($u = sy$) and a suffix of u (s and u are both suffixes of p and u is longer than s). Therefore, $s \prec u$. Define

$$U = \{u \mid u \text{ is a suffix of } \ p \ \wedge \ s \prec u\}$$

We have thus shown that there is a one-to-one correspondence between the elements of R'' and U. Given that $p = xsy$, $r = xs$, and $u = sy$, we have $m - \overline{r} = \overline{y} = \overline{u} - \overline{s}$. Hence,

$$\min(m - R'') \;=\; \min(U - \overline{s}).$$

For a fixed s, the minimum value of $\overline{u} - \overline{s}$ over all u in U is achieved for the shortest string v of U. We show that $c(v) = s$. This proves the result in (P4).

$$\begin{array}{ll} & v \text{ is the shortest string in } U \\ \Rightarrow & \{\overline{c(v)} < \overline{v}\} \\ & v \in U \;\wedge\; c(v) \notin U \\ \Rightarrow & \{\text{definition of } U\} \\ & s \prec v \;\wedge\; (c(v) \text{ is not a suffix of } \; p \;\vee\; \neg(s \prec c(v)) \\ \Rightarrow & \{c(v) \text{ is a suffix of } v \text{ and } v \text{ is a suffix of } p\text{; so, } c(v) \text{ is a suffix of } p\} \\ & s \prec v \;\wedge\; \neg(s \prec c(v)) \\ \Rightarrow & \{\text{definition of core}\} \\ & (s = c(v) \;\vee\; s \prec c(v)) \;\wedge\; (\neg(s \prec c(v))) \\ \Rightarrow & \{\text{predicate calculus}\} \\ & s = c(v) \end{array}$$ □

Note: The converse of this result is not true. There may be several u in U for which $c(u) = s$. For example, consider "sxs" and "sxyx", where the symbols 'x' and 'y' do not appear in "s". Cores for both of these strings are "s". □

An abstract program for computing $b(s)$

We derive a formula for $b(s)$, and use that to develop an abstract program.

$$\begin{array}{ll} & b(s) \\ = & \{\text{definition of } b(s) \text{ from Section 7.4.3}\} \\ & \min(m - R) \\ = & \{\text{from (P2): } \underline{c(p) \in R}\} \\ & \min(m - \overline{c(p)},\; \min(m - R)) \\ = & \{R = R' \cup R'', \text{ from Section 7.4.3}\} \\ & \min(m - \overline{c(p)},\; \min(m - R'),\; \min(m - R'')) \\ = & \{\text{from (P3): } \underline{\min(m - R') \;\geq\; m - \overline{c(p)}}\} \\ & \min(m - \overline{c(p)},\; \min(m - R'')) \\ = & \{\text{from (P4): } \underline{\min(m - R'') \;=\; \min(V - \overline{s})}\} \\ & \min(m - \overline{c(p)},\; \min(V - \overline{s})) \end{array}$$

Recall that

$$V = \{v \mid v \text{ is a suffix of } \; p \;\wedge\; c(v) = s\}$$

Now, we propose an abstract program to compute $b(s)$, for *all* suffixes s of p. We employ an array b where $b[s]$ ultimately holds the value of $b(s)$, though it is assigned different values during the computation. Initially, set $b[s]$ to $m - \overline{c(p)}$. Next, scan the suffixes v of p: let $s = c(v)$; update $b[s]$ to $\overline{v} - \overline{s}$ provided this value is lower than the current value of $b[s]$.

The program

assign $m - \overline{c(p)}$ to all elements of b;
for all suffixes v of p **do**
  $s := c(v)$;
  if $b[s] > \overline{v} - \overline{s}$ **then** $b[s] := \overline{v} - \overline{s}$ **endif**
endfor

A concrete program for computing $b(s)$

The goal of the concrete program is to compute an array e, where $e[j]$ is the amount by which the pattern is to be shifted when the matched suffix is $p[j..m]$, $0 \leq j \leq m$. That is,

$e[j] = b[s]$, where $j + \overline{s} = m$, or
$e[m - \overline{s}] = b[s]$, for any suffix s of p

We have no need to keep explicit prefixes and suffixes; instead, we keep their lengths, $\overline{s}$ in i and $\overline{v}$ in j. Let array f hold the lengths of the cores of all suffixes of p. Summarizing, for suffixes s and v of p,

$i = \overline{s}$,
$j = \overline{v}$,
$e[m - i] = b[s]$, using $i = \overline{s}$,
$f[\overline{v}] = \overline{c(v)}$, i.e., $f[j] = \overline{c(v)}$

The abstract program, given earlier, is transformed to the following concrete program.

assign $m - \overline{c(p)}$ to all elements of e;
for j, $0 \leq j \leq m$, **do**
  $i := f[j]$;
  if $e[m - i] > j - i$ **then** $e[m - i] := j - i$ **endif**
endfor

Computation of f The given program is complete except for the computation of f, the lengths of the cores of the suffixes of p. We have already developed a program to compute the cores of the prefixes of a string; we employ that program to compute f, as described next.

For any string r, let $\hat{r}$ be its reverse. Now, v is a suffix of p iff $\hat{v}$ is a prefix of $\hat{p}$. Moreover for any r (see exercise below)

$$c(\hat{r}) = \widehat{c(r)}$$

Therefore, for any suffix v of p and $u = \hat{v}$,

$c(u) = \widehat{c(v)}$, replace r by v, above; note: $\hat{r} = \hat{v} = u$
$\overline{c(v)} = \overline{\widehat{c(v)}}$, $\overline{r} = \overline{\hat{r}}$, for any r; let $r = c(v)$
$\overline{c(u)} = \overline{c(v)}$, from the above two

Since our goal is to compute the lengths of the cores, $\overline{c(v)}$, we compute $\overline{c(u)}$ instead, i.e., the lengths of the cores of the prefixes of $\hat{p}$, and store them in f.

Exercise 79

Show that

1. $r \preceq s \;\equiv\; \hat{r} \preceq \hat{s}$
2. $r \prec s \;\equiv\; \hat{r} \prec \hat{s}$
3. $c(\hat{r}) = \widehat{c(r)}$

Solution

1. $$\begin{array}{ll} & r \preceq s \\ \equiv & \{\text{definition of } \preceq\} \\ & r \text{ is a prefix of } s \text{ and } r \text{ is a suffix of } s \\ \equiv & \{\text{properties of prefix, suffix and reverse}\} \\ & \hat{r} \text{ is a suffix of } \hat{s} \text{ and } \hat{r} \text{ is a prefix of } \hat{s} \\ \equiv & \{\text{definition of } \preceq\} \\ & \hat{r} \preceq \hat{s} \end{array}$$

2. Similarly.

3. Indirect proof of equality is a powerful method for proving equality. This can be applied to elements in a set which has a reflexive and antisymmetric relation like $\preceq$. To prove $y = z$ for specific elements y and z, show that for every element x,

   $$x \preceq y \;\equiv\; x \preceq z.$$

   Then, set x to y to get $y \preceq y \;\equiv\; y \preceq z$, or $y \preceq z$ since $\preceq$ is reflexive. Similarly, get $z \preceq y$. Next, use antisymmetry of $\preceq$ to get $y = z$.

   We apply this method to prove the given equality: we show that for any s, $s \preceq c(\hat{r}) \;\equiv\; s \preceq \widehat{c(r)}$.

   $$\begin{array}{ll} & s \preceq c(\hat{r}) \\ \equiv & \{\text{definition of core}\} \\ & s \prec \hat{r} \\ \equiv & \{\text{second part of this exercise}\} \\ & \hat{s} \prec r \\ \equiv & \{\text{definition of core}\} \\ & \hat{s} \preceq c(r) \\ \equiv & \{\text{first part of this exercise}\} \\ & s \preceq \widehat{c(r)} \end{array}$$ □

Execution time for the computation of b The computation of $b(s)$, for all suffixes s of p, requires (1) computing $\overline{c(p)}$, (2) computing array f, and (3) executing the concrete program of this section. Note that (1) can be computed from array f; so, the steps (1,2) can be combined. The execution times of (1), (2) and (3) are linear in m, the length of p, from the text of the concrete program. So, array b can be computed in time that is linear in the length of the pattern.

Chapter 8

Parallel Recursion

8.1 Parallelism and Recursion

Many important synchronous parallel algorithms—Fast Fourier Transform, routing and permutation, Batcher sorting schemes, solving tridiagonal linear systems by odd-even reduction, prefix-sum algorithms—are conveniently formulated in a recursive fashion. The network structures on which parallel algorithms are typically implemented—butterfly, sorting networks, hypercube, complete binary tree—are, also, recursive in nature. However, parallel recursive algorithms are typically described iteratively, one parallel step at a time[1]. Similarly, the connection structures are often explained pictorially, by displaying the connections between one "level" and the next. The mathematical properties of the algorithms and connection structures are rarely evident from these descriptions.

A data structure, *powerlist*, is proposed in this paper that highlights the role of both parallelism and recursion. Many of the known parallel algorithms—FFT, Batcher Merge, prefix sum, embedding arrays in hypercubes, etc.—have surprisingly concise descriptions using powerlists. Simple algebraic properties of powerlists permit us to deduce properties of these algorithms employing structural induction.

8.2 Powerlist

The basic data structure on which recursion is employed (in LISP[34] or ML[35]) is a *list*. A list is either empty or it is constructed by concatenating an element to a list. (We restrict ourselves to finite lists throughout this paper.) We call such a list *linear* (because the list length grows by 1 as a result of applying the basic constructor). Such a list structure seems unsuitable for expressing parallel algorithms succinctly; an algorithm that processes the list elements has to describe how successive elements of the list are processed.

[1] A notable exception is the recursive description of a prefix sum algorithm in [23].

We propose *powerlist* as a data structure that is more suitable for describing parallel algorithms. The base—corresponding to the empty list for the linear case—is a list of one element. A longer powerlist is constructed from the elements of two powerlists of the same length, as described below. Thus, a powerlist is multiplicative in nature; its length doubles by applying the basic constructor.

There are two different ways in which powerlists are joined to create a longer powerlist. If p, q are powerlists of the same length then

$p \mid q$ is the powerlist formed by concatenating p and q, and

$p \bowtie q$ is the powerlist formed by successively taking alternate items from p and q, starting with p.

Further, we restrict p, q to contain similar elements (defined in Section 8.2.1).

In the following examples the sequence of elements of a powerlist are enclosed within angular brackets.

$\langle 0 \rangle \mid \langle 1 \rangle = \langle 0\ 1 \rangle$
$\langle 0 \rangle \bowtie \langle 1 \rangle = \langle 0\ 1 \rangle$
$\langle 0\ 1 \rangle \mid \langle 2\ 3 \rangle = \langle 0\ 1\ 2\ 3 \rangle$
$\langle 0\ 1 \rangle \bowtie \langle 2\ 3 \rangle = \langle 0\ 2\ 1\ 3 \rangle$

The operation $\mid$ is called *tie* and $\bowtie$ is *zip*.

8.2.1 Definitions

A data item from the linear list theory will be called a *scalar*. (Typical scalars are the items of base types—integer, boolean, etc.—tuples of scalars, functions from scalars to scalars and linear lists of scalars.) Scalars are uninterpreted in our theory. We merely assume that scalars can be checked for type compatibility. We will use several standard operations on scalars for purposes of illustration.

Notational Convention : Linear lists will be enclosed within square brackets, [ ].

A *powerlist* is a list of length 2^n, for some n, $n \geq 0$, all of whose elements are similar. We enclose powerlists within angular brackets, $\langle\ \rangle$.

Two scalars are *similar* if they are of the same type. Two powerlists are *similar* if they have the same length and any element of one is similar to any element of the other. (Observe that *similar* is an equivalence relation.)

Let S denote an arbitrary scalar, P a powerlist and u, v similar powerlists. A recursive definition of a powerlist is

$\langle S \rangle$ or $\langle P \rangle$ or $u \mid v$ or $u \bowtie v$

Examples

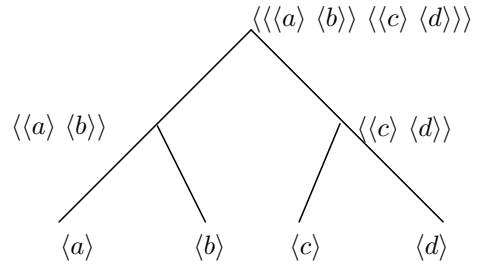


Figure 8.1: Representation of a complete binary tree where the data are at the leaves. For leaf nodes, the powerlist has one element. For non-leaf nodes, the powerlist has two elements, namely, the powerlists for the left and right subtrees.

⟨2⟩ powerlist of length 1 containing a scalar
⟨⟨2⟩⟩ powerlist of length 1 containing a powerlist of length 1 of scalar
⟨ ⟩ not a powerlist
⟨[ ]⟩ powerlist of length 1 containing the empty linear list
⟨ ⟨[2] [3 4 7]⟩ ⟨[4] [ ]⟩ ⟩
powerlist of length 2, each element of which is a powerlist of length 2, whose elements are linear lists of numbers
⟨ ⟨0 4⟩ ⟨1 5⟩ ⟨2 6⟩ ⟨3 7⟩ ⟩
a representation of the matrix $\begin{bmatrix} 0\ 1\ 2\ 3 \\ 4\ 5\ 6\ 7 \end{bmatrix}$ where each column is an element of the outer powerlist.
⟨ ⟨0 1 2 3⟩ ⟨4 5 6 7⟩ ⟩
another representation of the above matrix where each row is an element of the outer powerlist.
⟨⟨⟨a⟩ ⟨b⟩⟩ ⟨⟨c⟩ ⟨d⟩⟩⟩
a representation of the tree in Figure 8.1. The powerlist contains two elements, one each for the left and right subtrees.

8.2.2 Functions over Powerlists

Convention : We write function application without parantheses where no confusion is possible. Thus, we write "$f\ x$" instead of "$f(x)$" and "$g\ x\ y$" instead of "$g(x, y)$". The constructors $\mid$ and $\bowtie$ have the same binding power and their binding power is lower than that of function application. Throughout this paper, S denotes a scalar, P a powerlist and x, y either scalar or powerlist. Typical names for powerlist variables are p, q, r, s, t, u, v. □

Functions over linear lists are typically defined by case analysis—a function is defined over the empty list and, recursively, over non-empty lists. Functions over powerlists are defined analogously. For instance, the following function, rev, reverses the order of the elements of the argument powerlist.

$rev\langle x\rangle = \langle x\rangle$
$rev(p \mid q) = (rev\ q) \mid (rev\ p)$

The case analysis, as for linear lists, is based on the length of the argument powerlist. We adopt the pattern matching scheme of ML[35] and Miranda[49][2] to *deconstruct* the argument list into its components, p and q, in the recursive case. Deconstruction, in general, uses the operators $\mid$ and $\bowtie$; see Section 8.3. In the definition of rev, we have used $\mid$ for deconstruction; we could have used $\bowtie$ instead and defined rev in the recursive case by

$$rev(p \bowtie q) = (rev\ q) \bowtie (rev\ p)$$

It can be shown, using the laws in Section 8.3, that the two proposed definitions of rev are equivalent and that

$$rev(rev\ P) = P$$

for any powerlist P.

Scalar Functions

Operations on scalars are outside our theory. Some of the examples in this paper, however, use scalar functions, particularly, addition and multiplication (over complex numbers) and *cons* over linear lists. A scalar function, f, has zero or more scalars as arguments and its value is a scalar. We coerce the application of f to a powerlist by applying f "pointwise" to the elements of the powerlist. For a scalar function f of one argument we define

$$f\langle x\rangle = \langle f\ x\rangle$$
$$f(p \mid q) = (f\ p) \mid (f\ q)$$

It can be shown that

$$f(p \bowtie q) = (f\ p) \bowtie (f\ q)$$

A scalar function that operates on two arguments will often be written as an infix operator. For any such function $\oplus$ and similar powerlists p, q, u, v, we have

$$\langle x\rangle\ \oplus\ \langle y\rangle = \langle x\ \oplus\ y\rangle$$
$$(p \mid q)\ \oplus\ (u \mid v) = (p\ \oplus\ u) \mid (q\ \oplus\ v)$$
$$(p \bowtie q)\ \oplus\ (u \bowtie v) = (p\ \oplus\ u) \bowtie (q\ \oplus\ v)$$

Thus, scalar functions commute with both $\mid$ and $\bowtie$.

Note : Since a scalar function is applied recursively to each element of a powerlist, its effect propagates through all "levels". Thus, + applied to matrices forms their elementwise sum. □

[2]Miranda is a trademark of Research Software Ltd.

8.2.3 Discussion

The base case of a powerlist is a singleton list, not an empty list. Empty lists (or, equivalent data structures) do not arise in the applications we have considered. For instance, in matrix algorithms the base case is a 1×1 matrix rather than an empty matrix, Fourier transform is defined for a singleton list (not the empty list) and the smallest hypercube has one node.

The recursive definition of a powerlist says that a powerlist is either of the form $u \bowtie v$ or $u \mid v$. In fact, every non-singleton powerlist can be written in either form in a unique manner (see Laws in Section 8.3). A simple way to view $p \mid q = L$ is that if the elements of L are indexed by n-bit strings in increasing numerical order (where the length of L is 2^n) then p is the sublist of elements whose highest bit of the index is 0 and q is the sublist with 1 in the highest bit of the index. Similarly, if $u \bowtie v = L$ then u is the sublist of elements whose lowest bit of the index is 0 and v's elements have 1 as the lowest bit of the index.

At first, it may seem strange to allow two different ways for constructing the same list—using tie or zip. As we see in this paper this causes no difficulty, and further, this flexibility is essential because many parallel algorithms—the Fast Fourier Transform being the most prominent—exploit both forms of construction.

We have restricted u, v in $u \mid v$ and $u \bowtie v$ to be similar. This restriction allows us to process a powerlist by recursive divide and conquer, where each division yields two halves that can be processed in parallel, by employing the same algorithm. (Square matrices, for instance, are often processed by quartering them. We will show how quartering, or quadrupling, can be expressed in our theory.) The similarity restriction allows us to define complete binary trees, hypercubes and square matrices that are not "free" structures.

The length of a powerlist is a power of 2. This restricts our theory somewhat. It is possible to design a more general theory eliminating this constraint; we sketch an outline in Section 8.6.

8.3 Laws

L0. For singleton powerlists, $\langle x \rangle, \langle y \rangle$
$\langle x \rangle \mid \langle y \rangle = \langle x \rangle \bowtie \langle y \rangle$

L1. (Dual Deconstruction)
For any non-singleton powerlist, P, there exist similar powerlists r, s, u, v such that
$P = r \mid s$ and $P = u \bowtie v$

L2. (Unique Deconstruction)
$(\langle x \rangle = \langle y \rangle) \equiv (x = y)$
$(p \mid q = u \mid v) \equiv (p = u \wedge q = v)$
$(p \bowtie q = u \bowtie v) \equiv (p = u \wedge q = v)$

L3. (Commutativity of $\mid$ and $\bowtie$)
$(p \mid q) \bowtie (u \mid v) = (p \bowtie u) \mid (q \bowtie v)$

These laws can be derived by suitably defining tie and zip, using the standard functions from the linear list theory. One possible strategy is to define tie as the concatenation of two equal length lists and then, use the Laws L0 and L3 as the definition of zip; Laws L1, L2 can be derived next. Alternatively, these laws may be regarded as axioms relating tie and zip.

Law L0 is often used in proving base cases of algebraic identities. Laws L1, L2 allow us to uniquely deconstruct a non-singleton powerlist using either $\mid$ or $\bowtie$. Law L3 is crucial. It is the only law relating the two construction operators, $\mid$ and $\bowtie$, in the general case. Hence, it is invariably applied in proofs by structural induction where both constructors play a role.

Inductive Proofs

Most proofs on powerlists are by induction on the length, depth or shape of the list. The length, len, of a powerlist is the number of elements in it. Since the length of a powerlist is a power of 2, the logarithmic length, lgl, is a more useful measure. Formally,

$lgl\langle x \rangle = 0$
$lgl(u \mid v) = 1 + (lgl\ u)$

The *depth* of a powerlist is the number of "levels" in it.

$depth\ \langle S \rangle = 0$
$depth\ \langle P \rangle = 1\ + (depth\ P)$
$depth\ (u \mid v) = depth\ u$

(In the last case, since u, v are similar powerlists they have the same depth.) Most inductive proofs on powerlists order them lexicographically on the pair (depth, logarithmic length). For instance, to prove that a property Π holds for all powerlists, it is sufficient to prove

$\Pi\langle S \rangle$, and
$\Pi\ P\ \Rightarrow\ \Pi\langle P \rangle$, and
$(\Pi\ u)\ \wedge\ (\Pi\ v)\ \wedge\ (u, v)$ similar $\Rightarrow\ \Pi(u \mid v)$

The last proof step could be replaced by

$(\Pi\ u)\ \wedge\ (\Pi\ v)\ \wedge\ (u, v)$ similar $\Rightarrow\ \Pi(u \bowtie v)$

The *shape* of a powerlist P is a sequence of natural numbers $n_0, n_1, \ldots, n_d$ where d is the depth of P and

n_0 is the logarithmic length of P,
n_1 is the logarithmic length of (any) element of P, say r
n_2 is the logarithmic length of any element of $r, \ldots$
$\vdots$

A formal definition of shape is similar to that of depth. The shape is a linear sequence because all elements, at any level, are similar. The shape and the type of the scalar elements define the structure of a powerlist completely. For inductive proofs, the powerlists may be ordered lexicographically by the pair (depth, shape), where the shapes are compared lexicographically.

Example : The *len*, *lgl* and *depth* of $\langle\ \langle 0\ 1\ 2\ 3\rangle\ \langle 4\ 5\ 6\ 7\rangle\ \rangle$ are, 2, 1, 1, respectively. The *shape* of this powerlist is the sequence, 1 2, because there are 2 elements at the outer level and 4 elements at the inner level.

8.4 Examples

We show a few small algorithms on powerlists. These include such well-known examples as the Fast Fourier Transform and Batcher sorting schemes. We restrict the discussion in this section to simple (unnested) powerlists (where the depth is 0); higher dimensional lists (and algorithms for matrices and hypercubes) are taken up in a later section. Since the powerlists are unnested, induction based on length is sufficient to prove properties of these algorithms.

8.4.1 Permutations

We define a few functions that permute the elements of powerlists. The function rev, defined in Section 8.2.2, is a permutation function. These functions appear as components of many parallel algorithms.

Rotate

Function rr rotates a powerlist to the right by one; thus, $rr\langle a\ b\ c\ d\rangle = \langle d\ a\ b\ c\rangle$. Function rl rotates to the left: $rl\langle a\ b\ c\ d\rangle = \langle b\ c\ d\ a\rangle$.

$$\begin{array}{lcl} rr\langle x\rangle = \langle x\rangle & , & rl\langle x\rangle = \langle x\rangle \\ rr(u \bowtie v) = (rr\ v) \bowtie u & , & rl(u \bowtie v) = v \bowtie (rl\ u) \end{array}$$

There does not seem to be any simple definition of rr or rl using $|$ as the deconstruction operator. It is easy to show, using structural induction, that rr, rl are inverses. An amusing identity is $rev(rr(rev(rr\ P))) = P$.

A powerlist may be rotated through an arbitrary amount, k, by applying k successive rotations. A better scheme for rotating $(u \bowtie v)$ by k is to rotate both u, v by about $k/2$. More precisely, the function *grr* (given below) rotates a powerlist to the right by k, where $k \geq 0$. It is straightforward to show that for all $k, k \geq 0$, and all p, $(grr\ k\ p) = (rr^{(k)}\ p)$, where $rr^{(k)}$ is the k-fold application of rr.

$$\begin{array}{l} grr\ k\ \langle x\rangle = \langle x\rangle \\ grr\ (2 \times k)\ (u \bowtie v) = (grr\ k\ u) \bowtie (grr\ k\ v) \\ grr\ (2 \times k + 1)\ (u \bowtie v) = (grr\ (k+1)\ v) \bowtie (grr\ k\ u) \end{array}$$

P's indices	=	(000	001	010	011	100	101	110	111)
List P	=	$\langle a$	b	c	d	e	f	g	$h\rangle$
P's indices rotated right	=	(000	100	001	101	010	110	011	111)
$rs\ P$	=	$\langle a$	c	e	g	b	d	f	$h\rangle$
P's indices rotated left	=	(000	010	100	110	001	011	101	111)
$ls\ P$	=	$\langle a$	e	b	f	c	g	d	$h\rangle$

Figure 8.2: Permutation functions rs, ls defined in Section 8.4.1.

Rotate Index

A class of permutation functions can be defined by the transformations on the element indices. For a powerlist of 2^n elements we associate an n-bit index with each element, where the indices are the binary representations of $0, 1, .., 2^n - 1$ in sequence. (For a powerlist $u \mid v$, indices for the elements in u have "0" as the highest bit and in v have "1" as the highest bit. In $u \bowtie v$, similar remarks apply for the lowest bit.) Any bijection, h, mapping indices to indices defines a permutation of the powerlist: The element with index i is moved to the position where it has index $(h\ i)$. Below, we consider two simple index mapping functions; the corresponding permutations of powerlists are useful in describing the shuffle-exchange network. Note that indices are not part of our theory.

A function that rotates an index to the right (by one position) has the permutation function rs (for *r*ight *s*huffle) associated with it. The definition of rs may be understood as follows. The effect of rotating an index to the right is that the lowest bit of an index becomes the highest bit; therefore, if rs is applied to $u \bowtie v$, the elements of u—those having 0 as the lowest bit—will occupy the first half of the resulting powerlist (because their indices have "0" as the highest bit, after rotation); similarly, v will occupy the second half. Analogously, the function that rotates an index to the left (by one position) induces the permutation defined by ls (for *l*eft *s*huffle), below. Figure 8.2 shows the effects of index rotations on an 8-element list.

$$rs\langle x\rangle = \langle x\rangle \quad , \quad ls\langle x\rangle = \langle x\rangle$$
$$rs(u \bowtie v) = u \mid v \quad , \quad ls(u \mid v) = u \bowtie v$$

It is trivial to see that rs, ls are inverses.

Inversion

The function inv is defined by the following function on indices. An element with index b in P has index b' in $(inv\ P)$, where b' is the reversal of the bit string b. Thus,

$$\begin{array}{rcccccccccc} & {\scriptstyle 000} & {\scriptstyle 001} & {\scriptstyle 010} & {\scriptstyle 011} & {\scriptstyle 100} & {\scriptstyle 101} & {\scriptstyle 110} & {\scriptstyle 111} & & \\ inv\langle & a & b & c & d & e & f & g & h & \rangle & = \\ \langle & a & e & c & g & b & f & d & h & \rangle & \end{array}$$

The definition of inv is

$$\begin{aligned} &\text{inv}\langle x\rangle = \langle x\rangle \\ &\text{inv}(p \mid q) = (inv\ p) \bowtie (inv\ q) \end{aligned}$$

This function arises in a variety of contexts. In particular, inv is used to permute the output of a Fast Fourier Transform network into the correct order.

The following proof shows a typical application of structural induction.

INV1. $\text{inv}(p \bowtie q) = (inv\ p) \mid (inv\ q)$

Proof is by structural induction on p and q.

Base : $inv(\langle x\rangle \bowtie \langle y\rangle)$

$=$ {From Law L0 : $\langle x\rangle \bowtie \langle y\rangle = \langle x\rangle \mid \langle y\rangle$}

$inv(\langle x\rangle \mid \langle y\rangle)$

$=$ {definition of inv}

$inv\langle x\rangle \bowtie inv\langle y\rangle$

$=$ {$inv\langle x\rangle = \langle x\rangle, inv\langle y\rangle = \langle y\rangle$. Thus, they are singletons. Applying Law L0}

$inv\langle x\rangle \mid inv\langle y\rangle$

Induction :

$inv((r \mid s) \bowtie (u \mid v))$

$=$ {commutativity of $\mid$, $\bowtie$ }

$inv((r \bowtie u) \mid (s \bowtie v))$

$=$ {definition of inv}

$inv(r \bowtie u) \bowtie inv(s \bowtie v)$

$=$ {induction}

$(inv\ r \mid inv\ u) \bowtie (inv\ s \mid inv\ v)$

$=$ { $\mid$, $\bowtie$ commute}

$(inv\ r \bowtie inv\ s) \mid (inv\ u \bowtie inv\ v)$

$=$ {apply definition of inv to both sides of $\mid$ }

$inv(r \mid s) \mid inv(u \mid v)$ □

Using INV1 and structural induction, it is easy to establish

$$inv(inv\ P) = P\ ,$$

$$inv(rev\ P) = rev(inv\ P)$$

$n = 0$ $\langle[\ ]\rangle$
$n = 1$ $\langle[0]\ [1]\rangle$
$n = 2$ $\langle[00]\ [01]\ [11]\ [10]\rangle$
$n = 3$ $\langle[000]\ [001]\ [011]\ [010]\ [110]\ [111]\ [101]\ [100]\rangle$

Figure 8.3: Standard Gray code sequence for n, $n = 0, 1, 2, 3$

and for any scalar operator $\oplus$

$$inv(P \oplus Q) = (inv\ P) \oplus (inv\ Q)$$

The last result holds for any permutation function in place of inv.

8.4.2 Reduction

In the linear list theory [5], reduction is a higher order function of two arguments, an associative binary operator and a list. Reduction applied to $\oplus$ and $[a_0 a_1 \ldots a_n]$ yields $(a_0 \oplus a_1 \oplus \ldots \oplus a_n)$. This function over powerlists is defined by

$$red \oplus \langle x \rangle = x$$
$$red \oplus (p \mid q) = (red \oplus p)\ \oplus\ (red \oplus q)$$

8.4.3 Gray Code

Gray code sequence [17] for n, $n \geq 0$, is a sequence of 2^n n-bit strings where the consecutive strings in the sequence differ in exactly one bit position. (The last and the first strings in the sequence are considered consecutive.) Standard Gray code sequences for $n = 0, 1, 2, 3$ are shown in Figure 8.3. We represent the n-bit strings by linear lists of length n and a Gray code sequence by a powerlist whose elements are these linear lists. The standard Gray code sequence may be computed by function G, for any n.

$$G\ 0 = \langle[\ ]\rangle$$
$$G\ (n+1) = (0 : P) \mid (1 : (rev\ P))$$
$$\text{where } P = (G\ n)$$

Here, $(0\ :)$ is a scalar function that takes a linear list as an argument and appends 0 as its prefix. According to the coercion rule, $(0 : P)$ is the powerlist obtained by prefixing every element of P by 0. Similarly, $(1 : (rev\ P))$ is defined, where the function rev is from Section 8.2.2.

8.4.4 Polynomial

A polynomial with coefficients $p_j,\ 0 \leq j < 2^n$, where $n \geq 0$, may be represented by a powerlist p whose j^{th} element is p_j. The polynomial value at some point ω is $\sum_{0 \leq j < 2^n} p_j \times \omega^j$. For $n > 0$ this quantity is

$$\sum_{0 \leq j < 2^{n-1}} p_{2j} \times \omega^{2j} + \sum_{0 \leq j < 2^{n-1}} p_{2j+1} \times \omega^{2j+1}.$$

The following function, ep, evaluates a polynomial p using this strategy. In anticipation of the Fast Fourier Transform, we generalize ep to accept an arbitrary powerlist as its second argument. For powerlists $p,\ w$ (of, possibly, unequal lengths) let $(p\ ep\ w)$ be a powerlist of the same length as w, obtained by evaluating p at each element of w.

$$\langle x \rangle\ ep\ w = \langle x \rangle$$
$$(p \bowtie q)\ ep\ w = (p\ ep\ w^2) + (w \times (q\ ep\ w^2))$$

Note that w^2 is the pointwise squaring of w. Also, note that ep is a pointwise function in its second argument, i.e.,

$$p\ ep\ (u \mid v) = (p\ ep\ u) \mid (p\ ep\ v)$$

8.4.5 Fast Fourier Transform

For a polynomial p with complex coefficients, its Fourier transform is obtained by evaluating p at a sequence (i.e., powerlist) of points, $(W\ p)$. Here, $(W\ p)$ is the powerlist $\langle \omega^0, \omega^1, \ .. \ , \omega^{n-1} \rangle$, where n is the length of p and ω is the n^{th} principal root of 1. Note that $(W\ p)$ depends only on the length of p but not its elements; hence, for similar powerlists $p,\ q$, $(W\ p) = (W\ q)$. It is easy to define the function W in a manner similar to ep.

We need the following properties of W for the derivation of FFT. Equation (1) follows from the definition of W and the fact that $\omega^{2 \times N} = 1$, where N is the length of p (and q). The second equation says that the right half of $W(p \bowtie q)$ is the negation of its left half. This is because each element in the right half is the same as the corresponding element in the left half multiplied by ω^N; since ω is the $(2 \times N)^{th}$ root of 1, $\omega^N = -1$.

$$W^2(p \bowtie q) = (W\ p) \mid (W\ q) \tag{8.1}$$
$$W(p \bowtie q) = u \mid (-u), \text{for some } u \tag{8.2}$$

The Fourier transform, FT, of a powerlist p is a powerlist of the same length as p, given by

$$FT\ p = p\ ep\ (W\ p)$$

where ep is the function defined in Section 8.4.4.

The straightforward computation of $(p\ ep\ v)$ for any p, v consists of evaluating p at each element of v; this takes time $O(N^2)$ where p, v have length

N. Since $(W\ p)$ is of a special form the Fourier transform can be computed in $O(N\ \log N)$ steps, using the the Fast Fourier Transform algorithm [12]. This algorithm also admits an efficient parallel implementation, requiring $O(\log N)$ steps on $O(N)$ processors. We derive the FFT algorithm next.

$$
\begin{array}{ll}
 & FT\langle x\rangle \\
= & \{\text{definition of } FT\} \\
 & x\ ep\ (W\langle x\rangle) \\
= & \{\text{Since } W\langle x\rangle \text{ is a singleton, from the definition of } ep\} \\
 & \langle x\rangle
\end{array}
$$

For the general case,

$$
\begin{array}{ll}
 & FT(p \bowtie q) \\
= & \{\text{From the definition of } FT\} \\
 & (p \bowtie q)\ ep\ W(p \bowtie q) \\
= & \{\text{from the definition of } ep\} \\
 & p\ ep\ W^2(p \bowtie q) + W(p \bowtie q) \times (q\ ep\ W^2(p \bowtie q)) \\
= & \{\text{from the property of } W\text{; see equation (1)}\} \\
 & p\ ep\ ((W\ p) \mid (W\ q)) + W(p \bowtie q) \times (q\ ep\ ((W\ p) \mid (W\ q))) \\
= & \{\text{distribute each } ep \text{ over its second argument}\} \\
 & ((p\ ep\ (W\ p)) \mid (p\ ep\ (W\ q))) + W(p \bowtie q) \times ((q\ ep\ (W\ p)) \mid (q\ ep\ (W\ q))) \\
= & \{(W\ p) = (W\ q),\ p\ ep\ (W\ p) = FT\ p,\ q\ ep\ (W\ q) = FT\ q\ \} \\
 & ((FT\ p) \mid (FT\ p)) + W(p \bowtie q) \times ((FT\ q) \mid (FT\ q)) \\
= & \{\text{using } P, Q \text{ for } FT\ p,\ FT\ q, \text{ and } u \mid (-u) \text{ for } W(p \bowtie q)\text{; see equation (2)}\} \\
 & (P \mid P) + (u \mid -u) \times (Q \mid Q) \\
= & \{\ \mid \text{ and } \times \text{ in the second term commute}\} \\
 & (P \mid P) + ((u \times Q) \mid (-u \times Q)) \\
= & \{\ \mid \text{ and } + \text{ commute}\} \\
 & (P + u \times Q) \mid (P - u \times Q)
\end{array}
$$

We collect the two equations for FT to define FFT, the Fast Fourier Transform. In the following, $(powers\ p)$ is the powerlist $\langle \omega^0, \omega^1, \ .. \ , \omega^{N-1}\rangle$ where N is the length of p and ω is the $(2 \times N)^{th}$ principal root of 1. This was the value of u in the previous paragraph. The function $powers$ can be defined similarly to ep.

$$
\begin{array}{lll}
FFT\langle x\rangle = \langle x\rangle & & \\
FFT(p \bowtie q) = & (P + u \times Q) \mid (P - u \times Q) & \\
\quad \text{where} & P = FFT\ p & \\
 & Q = FFT\ q & \\
 & u = \ powers\ p &
\end{array}
$$

It is clear that $FFT(p \bowtie q)$ can be computed from $(FFT\ p)$ and $(FFT\ q)$ in $O(N)$ sequential steps or $O(1)$ parallel steps using $O(N)$ processors (u can be

computed in parallel), where N is the length of p. Therefore, $FFT(p \bowtie q)$ can be computed in $O(N \log N)$ sequential steps or, $O(\log N)$ parallel steps using $O(N)$ processors.

The compactness of this description of FFT is in striking contrast to the usual descriptions; for instance, see [10, Section 6.13]. The compactness can be attributed to the use of recursion and the avoidance of explicit indexing of the elements by employing $\mid$ and $\bowtie$. FFT illustrates the need for including both $\mid$ and $\bowtie$ as constructors for powerlists. (Another function that employs both $\mid$ and $\bowtie$ is inv of Section 8.4.1.)

Inverse Fourier Transform

The inverse of the Fourier Transform, IFT, can be defined similarly to the FFT. We derive the definition of IFT from that of the FFT by pattern matching.

For a singleton powerlist, $\langle x \rangle$, we compute

$$\begin{array}{ll} & IFT\langle x\rangle \\ = & \quad \{\langle x\rangle = FFT\langle x\rangle\} \\ & IFT(FFT\langle x\rangle) \\ = & \quad \{\text{IFT, FFT are inverses}\} \\ & \langle x\rangle \end{array}$$

For the general case, we have to compute $IFT(r \mid s)$ given r, s. Let

$$IFT(r \mid s) = p \bowtie q$$

in the unknowns p, q. This form of deconstruction is chosen so that we can easily solve the equations we generate, next. Taking FFT of both sides,

$$FFT(IFT(r \mid s)) = FFT(p \bowtie q)$$

The left side is $(r \mid s)$ because IFT, FFT are inverses. Replacing the right side by the definition of $FFT(p \bowtie q)$ yields the following equations.

$$\begin{array}{l} r \mid s = (P + u \times Q) \mid (P - u \times Q) \\ P = FFT\ p \\ Q = FFT\ q \\ u = \ powers\ p \end{array}$$

These equations are easily solved for the unknowns P, Q, u, p, q. (The law of unique deconstruction, L2, can be used to deduce from the first equation that $r = P + u \times Q$ and $s = P - u \times Q$. Also, since p and r are of the same length we may define u using r instead of p.) The solutions of these equations yield the following definition for IFT. Here, /2 divides each element of the given powerlist by 2.

$$\begin{array}{l} IFT\langle x\rangle = \langle x\rangle \\ IFT(r \mid s) = p \bowtie q \\ \quad \text{where} \quad P = (r + s)/2 \end{array}$$

$$u = powers\ r$$
$$Q = ((r - s)/2)/u$$
$$p = IFT\ P$$
$$q = IFT\ Q$$

As in the FFT, the definition of IFT includes both constructors, $|$ and $\bowtie$. It can be implemented efficiently on a butterfly network. The complexity of IFT is same as that of the FFT.

8.4.6 Batcher Sort

In this section, we develop some elementary results about sorting and discuss two remarkable sorting methods due to Batcher[4]. We find it interesting that $\bowtie$ (not $|$) is the preferred operator in discussing the principles of parallel sorting. Henceforth, a list is *sorted* means that its elements are arranged in non-decreasing order.

A general method of sorting is given by

$$sort\langle x\rangle = \langle x\rangle$$
$$sort(p \bowtie q) = (sort\ p)\ merge\ (sort\ q)$$

where *merge* (written as a binary infix operator) creates a single sorted powerlist out of the elements of its two argument powerlists each of which is sorted. In this section, we show two different methods for implementing *merge*. One scheme is Batcher merge, given by the operator *bm*. Another scheme is given by *bitonic* sort where the sorted lists u, v are merged by applying the function *bi* to $(u \mid (rev\ v))$.

A comparison operator, $\updownarrow$, is used in these algorithms. The operator is applied to a pair of equal length powerlists, p, q; it creates a single powerlist out of the elements of p, q by

$$p \updownarrow q = (p\ \min\ q) \bowtie (p\ \max\ q)$$

That is, the $2i^{th}$ and $(2i+1)^{th}$ items of $p \updownarrow q$ are $(p_i\ \min\ q_i)$ and $(p_i\ \max\ q_i)$, respectively. The powerlist $p\ \updownarrow\ q$ can be computed in constant time using $O(len\ p)$ processors.

Bitonic Sort

A sequence of numbers, $x_0, x_1, .., x_i, .., x_N$, is *bitonic* if there is an index i, $0 \leq i \leq N$, such that $x_0, x_1, .., x_i$ is monotonic (ascending or descending) and $x_i, .., x_N$ is monotonic. The function *bi*, given below, applied to a bitonic powerlist returns a sorted powerlist of the original items.

$$bi\langle x\rangle = \langle x\rangle$$
$$bi(p \bowtie q) = (bi\ p) \updownarrow (bi\ q)$$

For sorted powerlists u, v, the powerlist $(u \mid (rev\ v))$ is bitonic; thus u, v can be merged by applying bi to $(u \mid (rev\ v))$. The form of the recursive definition suggests that bi can be implemented on $O(N)$ processors in $O(\log\ N)$ parallel steps, where N is the length of the argument powerlist.

Batcher Merge

Batcher has also proposed a scheme for merging two sorted lists. We define this scheme, bm, as an infix operator below.

$$\langle x\rangle\ bm\ \langle y\rangle = \langle x\rangle \updownarrow \langle y\rangle$$
$$(r \bowtie s)\ bm\ (u \bowtie v) = (r\ bm\ v) \updownarrow (s\ bm\ u)$$

The function bm is well-suited for parallel implementation. The recursive form suggests that $(r\ bm\ v)$ and $(s\ bm\ u)$ can be computed in parallel. Since $\updownarrow$ can be applied in $O(1)$ parallel steps using $O(N)$ processors, where N is the length of the argument powerlists, the function bm can be evaluated in $O(\log N)$ parallel steps. In the rest of this section, we develop certain elementary facts about sorting and prove the correctness of bi and bm.

Elementary Facts about Sorting

We consider only "compare and swap" type sorting methods. It is known (see [26]) that such a sorting scheme is correct if and only if it sorts lists containing 0's and 1's only. Therefore, we restrict our discussion to powerlists containing 0's and 1's, only.

For a powerlist p, let $(z\ p)$ be the number of 0's in it. To simplify notation, we omit the space and write zp. Clearly,

A0. $z(p \bowtie q) = zp + zq$ and $z\langle x\rangle$ is either 0 or 1.

Powerlists containing only 0's and 1's have the following properties.

A1. $\langle x\rangle$ sorted and $\langle x\rangle$ bitonic.
A2. $(p \bowtie q)$ sorted $\equiv$ p sorted $\wedge$ q sorted $\wedge\ 0 \leq zp - zq \leq 1$
A3. $(p \bowtie q)$ bitonic $\Rightarrow$ p bitonic $\wedge$ q bitonic $\wedge\ |zp - zq| \leq 1$

Note : The condition analogous to (A2) under which $p \mid q$ is sorted is,

A2′. $(p \mid q)$ sorted $\equiv$ p sorted $\wedge$ q sorted $\wedge\ (zp < (len\ p)\ \Rightarrow\ zq = 0)$

The simplicity of (A2), compared with (A2′), may suggest why $\bowtie$ is the primary operator in parallel sorting. □

The following results, (B1, B2), are easy to prove. We prove (B3).

B1. p sorted, q sorted, $zp \geq zq\ \Rightarrow\ (p\ \min\ q) = p\ \wedge\ (p\ \max\ q) = q$ □

B2. $z(p \updownarrow q) = zp + zq$ □

B3. p sorted, q sorted, $|zp - zq| \leq 1 \;\Rightarrow\; (p \updownarrow q)$ sorted

Proof: Since the statement of B3 is symmetric in p, q, assume $zp \geq zq$.

p sorted, q sorted, $|zp - zq| \leq 1$
$\Rightarrow$ {assumption: $zp \geq zq$}
p sorted, q sorted, $0 \leq zp - zq \leq 1$
$\Rightarrow$ {A2 and B1}
$p \bowtie q$ sorted, $(p \;\min\; q) = p, (p \;\max\; q) = q$
$\Rightarrow$ {replace p, q in $p \bowtie q$ by $(p \;\min\; q), (p \;\max\; q)$}
$(p \;\min\; q) \bowtie (p \;\max\; q)$ sorted
$\Rightarrow$ {definition of $p \updownarrow q$}
$p \updownarrow q$ sorted

Correctness of Bitonic Sort

We show that the function *bi* applied to a bitonic powerlist returns a sorted powerlist of the original elements: (B4) states that *bi* preserves the number of zeroes of its argument list (i.e., it loses no data) and (B5) states that the resulting list is sorted.

B4. $z(bi\ p) = zp$

Proof: By structural induction, using B2. □

B5. L bitonic $\Rightarrow (bi\ L)$ sorted

Proof: By structural induction.

Base: Straightforward.

Induction: Let $L = p \bowtie q$
$p \bowtie q$ bitonic
$\Rightarrow$ {A3}
p bitonic, q bitonic, $|zp - zq| \leq 1$
$\Rightarrow$ {induction on p and q}
$(bi\ p)$ sorted, $(bi\ q)$ sorted, $|zp - zq| \leq 1$
$\Rightarrow$ {from B4: $z(bi\ p) = zp,\ z(bi\ q) = zq$}
$(bi\ p)$ sorted, $(bi\ q)$ sorted, $|z(bi\ p) - z(bi\ q)| \leq 1$
$\Rightarrow$ {apply B3 with $(bi\ p), (bi\ q)$ for p, q}
$(bi\ p) \updownarrow (bi\ q)$ sorted
$\Rightarrow$ {definition of *bi*}
$bi(p \bowtie q)$ sorted

Correctness of Batcher Merge

We can show that bm merges two sorted powerlists in a manner similar to the proof of bi. Instead, we establish a simple relationship between the functions bm and bi from which the correctness of the former is obvious. We show that

B6. $p\ bm\ q = bi(p \mid (rev\ q))$, where rev reverses a powerlist (Section 8.2.2).

If p, q are sorted then $p \mid (rev\ q)$ is bitonic (a fact that we don't prove here). Then, from the correctness of bi it follows that $bi(p \mid (rev\ q))$ and, hence, $p\ bm\ q$ is sorted (and it contains the elements of p and q).

Proof of B6: By structural induction.

Base: Let $p,\ q = \langle x\rangle,\ \langle y\rangle$

$$
\begin{aligned}
& bi(\langle x\rangle \mid rev\langle y\rangle) \\
= \quad & \{\text{definition of } rev\} \\
& bi(\langle x\rangle \mid \langle y\rangle) \\
= \quad & \{(\langle x\rangle \mid \langle y\rangle) = (\langle x\rangle \bowtie \langle y\rangle)\} \\
& bi(\langle x\rangle \bowtie \langle y\rangle) \\
= \quad & \{\text{definition of } bi\} \\
& \langle x\rangle \updownarrow \langle y\rangle \\
= \quad & \{\text{definition of } bm\} \\
& \langle x\rangle\ bm\ \langle y\rangle
\end{aligned}
$$

Induction: Let $p,\ q = r \bowtie s,\ u \bowtie v$

$$
\begin{aligned}
& bi(p \mid (rev\ q)) \\
= \quad & \{\text{expanding } p, q\} \\
& bi((r \bowtie s) \mid rev(u \bowtie v)) \\
= \quad & \{\text{definition of } rev\} \\
& bi((r \bowtie s) \mid (rev\ v \bowtie rev\ u)) \\
= \quad & \{\mid,\ \bowtie \text{ commute}\} \\
& bi((r \mid rev\ v) \bowtie (s \mid rev\ u)) \\
= \quad & \{\text{definition of } bi\} \\
& bi(r \mid rev\ v) \updownarrow bi(s \mid rev\ u) \\
= \quad & \{\text{induction}\} \\
& (r\ bm\ v) \updownarrow (s\ bm\ u) \\
= \quad & \{\text{definition of } bm\} \\
& (r \bowtie s)\ bm\ (u \bowtie v)
\end{aligned}
$$

$=$ {using the definitions of p, q}

$p\ bm\ q$ □

The compactness of the description of Batcher's sorting schemes and the simplicity of their correctness proofs demonstrate the importance of treating recursion and parallelism simultaneously.

8.4.7 Prefix Sum

Let L be a powerlist of scalars and $\oplus$ be a binary, associative operator on that scalar type. The prefix sum of L with respect to $\oplus$, $(ps\ L)$, is a list of the same length as L given by

$$ps\ \langle x_0, x_1, .., x_i, .., x_N\rangle = \langle x_0, x_0 \oplus x_1, .., x_0 \oplus x_1 \oplus ..x_i, .., x_0 \oplus x_1 \oplus .. \oplus x_N\rangle,$$

that is, in $(ps\ L)$ the element with index i, $i > 0$, is obtained by applying $\oplus$ to the first $(i+1)$ elements of L in order. We will give a formal definition of prefix sum later in this section.

Prefix sum is of fundamental importance in parallel computing. We show that two known algorithms for this problem can be concisely represented and proved in our theory. Again, zip turns out to be the primary operator for describing these algorithms.

A particularly simple scheme for prefix sum of 8 elements is shown in Figure 8.4. In that figure, the numbered nodes represent processors, though the same 8 physical processors are used at all levels. Initially, processor i holds the list element L_i, for all i. The connections among the processors at different levels depict data transmissions. In level 0, each processor, from 0 through 6, sends its data to its right neighbor. In the i^{th} level, processor i sends its data to $(i + 2^i)$, if such a processor exists (this means that for $j < 2^i$, processor j receives no data in level i data transmission). Each processor updates its own data, d, to $r \oplus d$ where r is the data it receives; if it receives no data in some level then d is unchanged. It can be shown that after completion of the computation at level $(\log_2(len\ L))$, processor i holds the i^{th} element of $(ps\ L)$.

Another scheme, due to Ladner and Fischer[29], first applies $\oplus$ to adjacent elements x_{2i}, x_{2i+1} to compute the list $\langle x_0 \oplus x_1, ..\ x_{2i} \oplus x_{2i+1}, ..\rangle$. This list has half as many elements as the original list; its prefix sum is then computed recursively. The resulting list is $\langle x_0 \oplus x_1, .., x_0 \oplus x_1 \oplus .. \oplus x_{2i} \oplus x_{2i+1}, \ldots\rangle$. This list contains half of the elements of the final list; the missing elements are $x_0, x_0 \oplus x_1 \oplus x_2, .., x_0 \oplus x_1 \oplus .. \oplus x_{2i}, ...$ These elements can be computed by "adding" $x_2, x_4, ..$, appropriately to the elements of the already computed list.

Both schemes for prefix computation are inherently recursive. Our formulations will highlight both parallelism and recursion.

Specification

As we did for the sorting schemes (Section 8.4.6), we introduce an operator in terms of which the prefix sum problem can be defined. First, we postulate

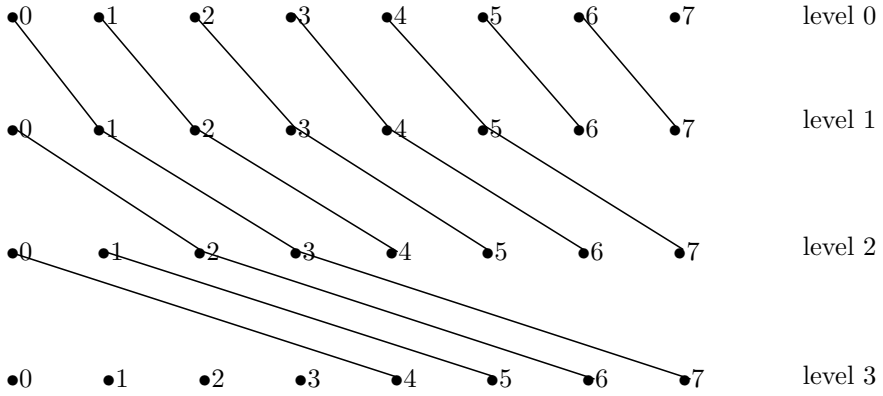


Figure 8.4: A network to compute the prefix sum of 8 elements.

that 0 is the left identity element of $\oplus$, i.e., $0 \oplus x = x$. For a powerlist p, let p^* be the powerlist obtained by shifting p to the right by one. The effect of shifting is to append a 0 to the left and discard the rightmost element of p; thus, $\langle a\ b\ c\ d\rangle^* = \langle 0\ a\ b\ c\rangle$. Formally,

$$\langle x\rangle^* = \langle 0\rangle$$
$$(p \bowtie q)^* = q^* \bowtie p$$

It is easy to show

$$S1. (r \oplus s)^* = r^* \oplus s^*$$
$$S2. (p \bowtie q)^{**} = p^* \bowtie q^*$$

Consider the following equation in the powerlist variable z.

$$z = z^* \oplus L \qquad \text{(DE)}$$

where L is some given powerlist. This equation has a unique solution in z, because

$$\begin{aligned} z_0 &= (z^*)_0 \oplus L_0 \\ &= 0 \oplus L_0 \\ &= L_0 \qquad\qquad \text{, and} \\ z_{i+1} &= z_i \oplus L_{i+1} \text{ , } 0 \le i < (len\ L) - 1 \end{aligned}$$

For $L = \langle a\ b\ c\ d\rangle$, $z = \langle a \quad a \oplus b \quad a \oplus b \oplus c \quad a \oplus b \oplus c \oplus d\rangle$ which is exactly $(ps\ L)$. *We define* $(ps\ L)$ *to be the unique solution of* (DE)*, and we call* (DE) *the defining equation for* $(ps\ L)$.

Notes

1. The operator $\oplus$ is not necessarily commutative. Therefore, the rhs of (DE) may not be the same as $L \oplus z^*$.

2. The operator $\oplus$ is scalar; so, it commutes with $\bowtie$.

3. The uniqueness of the solution of (DE) can be proved entirely within the powerlist algebra, similar to the derivation of Ladner-Fischer scheme given later in this section.

4. Adams[1] has specified the prefix-sum problem without postulating an explicit "0" element. For any $\oplus$, he introduces a binary operator $\vec{\oplus}$ over two similar powerlists such that $p\vec{\oplus}\ q = p^* \oplus q$. The operator $\vec{\oplus}$ can be defined without introducing a "0".

Computation of the Prefix Sum

The function *sps* (*s*imple *p*refix *s*um) defines the scheme of Figure 8.4.

$$
\begin{array}{l}
sps\ \langle x\rangle = \langle x\rangle \\
sps\ L = (sps\ u) \bowtie (sps\ v) \\
\qquad \text{where } u \bowtie v = L^* \oplus L
\end{array}
$$

In the first level in Figure 8.4, $L^* \oplus L$ is computed. If $L = \langle x_0, x_1, .., x_i, \ldots\rangle$ then this is $\langle x_0, x_0 \oplus x_1, .., x_i \oplus x_{i+1}..\rangle$. This is the zip of the two sublists $\langle x_0, x_1 \oplus x_2, .., x_{2i-1} \oplus x_{2i}, ..\rangle$ and $\langle x_0 \oplus x_1, .., x_{2i} \oplus x_{2i+1}, ..\rangle$. Next, prefix sums of these two lists are computed (independently) and then zipped.

The Ladner-Fischer scheme is defined by the function $\langle\forall$.

$$
\begin{array}{l}
\langle\forall\langle x\rangle = \langle x\rangle \\
\langle\forall(p \bowtie q) = (t^* \oplus p) \bowtie t \\
\qquad \text{where } t = \langle\forall(p \oplus q)
\end{array}
$$

Correctness

We can prove the correctness of *sps* and $\langle\forall$ by showing that the function *ps* satisfies the equations defining each of these functions. It is more instructive to see that both *sps* and $\langle\forall$ can be derived easily from the specification (DE). We carry out this derivation for the Fischer-Ladner scheme as an illustration of the power of algebraic manipulations. First, we note, $ps\langle x\rangle = \langle x\rangle$.

$$
\begin{array}{ll}
 & ps\langle x\rangle \\
= & \quad \{\text{from the defining equation DE for } ps\langle x\rangle\} \\
 & (ps\langle x\rangle)^* \oplus \langle x\rangle \\
= & \quad \{\text{definition of } ^*\} \\
 & \langle 0\rangle \oplus \langle x\rangle \\
= & \quad \{\oplus \text{ is a scalar operation}\} \\
 & \langle 0 \oplus x\rangle \\
= & \quad \{0 \text{ is the identity of } \oplus\} \\
 & \langle x\rangle
\end{array}
$$

Derivation of Ladner-Fischer Scheme

Given a powerlist $p \bowtie q$, we derive an expression for $ps(p \bowtie q)$. Let $r \bowtie t$, in unknowns r, t, be $ps(p \bowtie q)$. We solve for r, t.

$$
\begin{array}{ll}
 & r \bowtie t \\
= & \quad \{r \bowtie t = ps\ (p \bowtie q).\ \text{Using (DE)}\} \\
 & (r \bowtie t)^* \oplus (p \bowtie q) \\
= & \quad \{(r \bowtie t)^* = t^* \bowtie r\} \\
 & (t^* \bowtie r) \oplus (p \bowtie q) \\
= & \quad \{\oplus,\ \bowtie\ \text{commute}\} \\
 & (t^* \oplus p) \bowtie (r \oplus q)
\end{array}
$$

Applying law L2 (unique deconstruction) to the equation $r \bowtie t = (t^* \oplus p) \bowtie (r \oplus q)$, we conclude that

LF1. $r = t^* \oplus p$, and

LF2. $t = r \oplus q$

Now, we eliminate r from (LF2) using (LF1) to get $t = t^* \oplus p \oplus q$. Using (DE) and this equation we obtain

LF3. $t = ps(p \oplus q)$

We summarize the derivation of $ps(p \bowtie q)$.

$$
\begin{array}{ll}
 & ps(p \bowtie q) \\
= & \quad \{\text{by definition}\} \\
 & r \bowtie t \\
= & \quad \{\ \text{Using (LF1) for } r\} \\
 & (t^* \oplus p) \bowtie t
\end{array}
$$

where t is defined by LF3. This is exactly the definition of the function $\langle \forall$ for a non-singleton powerlist. We also note that

$$
\begin{array}{ll}
 & r \\
= & \quad \{\text{by eliminating } t \text{ from (LF1) using (LF2) }\} \\
 & (r \oplus q)^* \oplus p \\
= & \quad \{\ \text{definition of } ^*\} \\
 & r^* \oplus q^* \oplus p
\end{array}
$$

Using (DE) and this equation we obtain LF4 that is used in proving the correctness of *sps*, next.

LF4. $r = ps(q^* \oplus p)$

Correctness of *sps*

We show that for a non-singleton powerlist L,

$ps\ L = (ps\ u) \bowtie (ps\ v)$, where $u \bowtie v = L^* \oplus L$.

Proof: Let $L = p \bowtie q$. Then

$$\begin{array}{ll} & ps\ L \\ = & \quad \{L = p \bowtie q\} \\ & ps(p \bowtie q) \\ = & \quad \{ps(p \bowtie q) = r \bowtie t, \text{ where } r, t \text{ are given by (LF4,LF3)}\} \\ & ps(q^* \oplus p) \bowtie ps(p \oplus q) \\ = & \quad \{\text{Letting } u = q^* \oplus p,\ v = p \oplus q\} \\ & (ps\ u) \bowtie (ps\ v) \end{array}$$

Now, we show that $u \bowtie v = L^* \oplus L$.

$$\begin{array}{ll} & u \bowtie v \\ = & \quad \{u = q^* \oplus p,\ v = p \oplus q\} \\ & (q^* \oplus p) \bowtie (p \oplus q) \\ = & \quad \{\oplus,\ \bowtie \text{ commute}\} \\ & (q^* \bowtie p) \oplus (p \bowtie q) \\ = & \quad \{\text{Apply the definition of } * \text{ to the first term}\} \\ & (p \bowtie q)^* \oplus (p \bowtie q) \\ = & \quad \{L = p \bowtie q\} \\ & L^* \oplus L \end{array}$$

Remarks. A more traditional way of describing a prefix sum algorithm, such as the simple scheme of Figure 8.4, is to explicitly name the quantities that are being computed, and establish relationships among them. Let y_{ij} be computed by the i^{th} processor at the j^{th} level. Then, for all i, j, $0 \leq i < 2^n$, $0 \leq j < n$, where n is the logarithmic length of the list,

$$y_{i0} = x_i, \text{ and}$$
$$y_{i,j+1} = \left\{ \begin{array}{lll} y_{i-2^j,j} & , & i \geq 2^j \\ 0 & , & i < 2^j \end{array} \right\} \oplus y_{ij}$$

The correctness criterion is

$$y_{in} = x_0 \oplus .. \oplus x_i$$

This description is considerably more difficult to manipulate. The parallelism in it is harder to see. The proof of correctness requires manipulations of indices: for this example, we have to show that for all i, j

$$y_{ij} = x_k \oplus .. \oplus x_i$$
where $k = \max(0, i - 2^j + 1)$.

The Ladner-Fischer scheme is even more difficult to specify in this manner. Algebraic methods are to be preferred for describing uniform operations on aggregates of data.

8.5 Higher Dimensional Arrays

A major part of parallel computing involves arrays of one or more dimensions. An array of m dimensions (dimensions are numbered 0 through $m-1$) is represented by a powerlist of depth $(m-1)$. Conversely, since powerlist elements are similar, a powerlist of depth $(m-1)$ may be regarded as an array of dimension m. For instance, a matrix of r rows and c columns may be represented as a powerlist of c elements, each element being a powerlist of length r storing the items of a column; conversely, the same matrix may be represented by a powerlist of r elements, each element being a powerlist of c elements.

In manipulating higher dimensional arrays we prefer to think in terms of array operations rather than operations on nested powerlists. Therefore, we introduce construction operators, analogous to $|$ and $\bowtie$, for tie and zip along any specified dimension. We use $|', \bowtie'$ for the corresponding operators in dimension 1, $|'', \bowtie''$ for the dimension 2, etc. The definitions of these operators are in Section 8.5.2; for the moment it is sufficient to regard $|'$ as the pointwise application of $|$ to the argument powerlists (and similarly, $\bowtie'$). Thus, for similar (power) matrices A, B that are stored columnwise (i.e., each element is a column), $A \mid B$ is the concatenation of A, B by rows and $A \mid' B$ is their concatenation by columns. Figure 8.5 shows applications of these operators on specific matrices.

Given these constructors we may define a matrix to be either

a singleton matrix $\langle\langle x \rangle\rangle$, or
$p \mid q$ where p, q are (similar) matrices, or
$u \mid' v$ where u, v are (similar) matrices.

Analogous definitions can be given for n-dimensional arrays. Observe that the length of each dimension is a power of 2. As we had in the case of a powerlist, the same matrix can be constructed in several different ways, say, first

$$A = \left\langle \begin{array}{cc} \wedge & \wedge \\ 2 & 4 \\ 3 & 5 \\ \vee & \vee \end{array} \right\rangle \qquad B = \left\langle \begin{array}{cc} \wedge & \wedge \\ 0 & 1 \\ 6 & 7 \\ \vee & \vee \end{array} \right\rangle$$

$$A \mid B = \left\langle \begin{array}{cccc} \wedge & \wedge & \wedge & \wedge \\ 2 & 4 & 0 & 1 \\ 3 & 5 & 6 & 7 \\ \vee & \vee & \vee & \vee \end{array} \right\rangle \qquad A \bowtie B = \left\langle \begin{array}{cccc} \wedge & \wedge & \wedge & \wedge \\ 2 & 0 & 4 & 1 \\ 3 & 6 & 5 & 7 \\ \vee & \vee & \vee & \vee \end{array} \right\rangle$$

$$A \mid' B = \left\langle \begin{array}{cc} \wedge & \wedge \\ 2 & 4 \\ 3 & 5 \\ 0 & 1 \\ 6 & 7 \\ \vee & \vee \end{array} \right\rangle \qquad A \bowtie' B = \left\langle \begin{array}{cc} \wedge & \wedge \\ 2 & 4 \\ 0 & 1 \\ 3 & 5 \\ 6 & 7 \\ \vee & \vee \end{array} \right\rangle$$

Figure 8.5: Applying $\mid, \bowtie, \mid', \bowtie'$ over matrices. Matrices are stored by columns. Typical matrix format is used for display, though each matrix is to be regarded as a powerlist of powerlists.

by constructing the rows and then the columns, or vice versa. We will show, in Section 8.5.2, that

$$(p \mid q) \mid' (u \mid v) = (p \mid' u) \mid (q \mid' v)$$

i.e., $\mid, \mid'$ commute.

Note : We could have defined a matrix using $\bowtie$ and $\bowtie'$ instead of $\mid$ and $\mid'$. As $\mid$ and $\bowtie$ are duals in the sense that either can be used to construct (or uniquely deconstruct) a powerlist, $\mid'$ and $\bowtie'$ are also duals, as we show in Section 8.5.2. Therefore, we will freely use all four construction operators for matrices. □

Example : (Matrix Transposition)

Let τ be a function that transposes matrices. From the definition of a matrix, we have to consider three cases in defining τ.

$\tau\langle\langle x\rangle\rangle = \langle\langle x\rangle\rangle$
$\tau(p \mid q) = (\tau\ p) \mid' (\tau\ q)$
$\tau(u \mid' v) = (\tau\ u) \mid (\tau\ v)$

The description of function τ, though straightforward, has introduced the possibility of an inconsistent definition. For a 2×2 matrix, for instance, either of the last two deconstructions apply, and it is not obvious that the same result is obtained independent of the order in which the rules are applied. We show that τ is a function.

We prove the result by structural induction. For a matrix of the form $\langle\langle x\rangle\rangle$, only the first deconstruction applies, and, hence, the claim holds. Next, consider

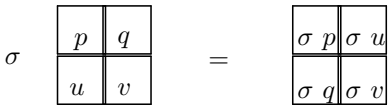


Figure 8.6: Schematic of the transposition of a square powermatrix.

a matrix to which both of the last two deconstructions apply. Such a matrix is of the form $(p \mid q) \mid' (u \mid v)$ which, as remarked above, is also $(p \mid' u) \mid (q \mid' v)$. Applying one step of each of the last two rules in different order, we get

$$
\begin{array}{ll}
 & \tau((p \mid q) \mid' (u \mid v)) \\
= & \quad \{\text{applying the last rule}\} \\
 & (\tau(p \mid q)) \mid (\tau(u \mid v)) \\
= & \quad \{\text{applying the middle rule}\} \\
 & ((\tau\ p) \mid' (\tau\ q)) \mid ((\tau\ u) \mid' (\tau\ v))
\end{array}
$$

And,

$$
\begin{array}{ll}
 & \tau((p \mid' u) \mid (q \mid' v)) \\
= & \quad \{\text{applying first the middle rule, then the last rule}\} \\
 & ((\tau\ p) \mid (\tau\ u)) \mid' ((\tau\ q) \mid (\tau\ v)) \\
= & \quad \{\mid, \mid' \text{ commute}\} \\
 & ((\tau\ p) \mid' (\tau\ q)) \mid ((\tau\ u) \mid' (\tau\ v))
\end{array}
$$

From the induction hypothesis, $(\tau\ p)$, $(\tau\ q)$, etc., are well defined. Hence,

$$\tau((p \mid q) \mid' (u \mid v)) = \tau((p \mid' u) \mid (q \mid' v))$$

Crucial to the above proof is the fact that $\mid$ and $\mid'$ commute; this is reminiscent of the "Church-Rosser Property" [11] in term rewriting systems. Commutativity is so important that we discuss it further in the next subsection.

It is easy to show that

$$\tau\ (p \bowtie q) = (\tau\ p) \bowtie' (\tau\ q) \text{ and}$$
$$\tau\ (u \bowtie' v) = (\tau\ u) \bowtie (\tau\ v)$$

Transposition of a square (power) matrix can be defined by deconstructing the matrix into quarters, transposing them individually and rearranging them, as shown in Figure 8.6. From the transposition function τ for general matrices, we get a function σ for transpositions of square matrices

$$\sigma\langle\langle x\rangle\rangle = \langle\langle x\rangle\rangle$$
$$\sigma((p \mid q) \mid' (u \mid v)) = ((\sigma\ p) \mid' (\sigma\ q)) \mid ((\sigma\ u) \mid' (\sigma\ v))$$

Note the effectiveness of pattern matching in this definition.

8.5.1 Pointwise Application

Let g be a function mapping items of type α to type β. Then g' maps a powerlist of α-items to a powerlist of β-items.

$$g'\langle x\rangle = \langle g\ x\rangle$$
$$g'(r \mid s) = (g'\ r) \mid (g'\ s)$$

Similarly, for a binary operator *op*

$$\langle x\rangle\ op'\ \langle y\rangle = \langle x\ op\ y\rangle$$
$$(r \mid s)\ op'\ (u \mid v) = (r\ op'\ u) \mid (s\ op'\ v)$$

We have defined these two forms explicitly because we use one or the other in all our examples; f' for a function f of arbitrary arity is similarly defined. Observe that f' applied to a powerlist of length N yields a powerlist of length N. The number of primes over f determines the dimension at which f is applied (the outermost dimension is numbered 0; therefore writing $\bowtie$, for instance, without primes, simply zips two lists). The operator for pointwise application also appears in [3] and in [46].

Common special cases for the binary operator, *op*, are $\mid$ and $\bowtie$ and their pointwise application operators. In particular, writing $\bowtie^m$ to denote $\bowtie\ \overbrace{''\cdots'}^{m}$, we define, $\bowtie^0 = \bowtie$ and for $m > 0$,

$$\langle x\rangle \bowtie^m \langle y\rangle = \langle x \bowtie^{m-1} y\rangle$$
$$(r \mid s) \bowtie^m (u \mid v) = (r \bowtie^m u) \mid (s \bowtie^m v)$$

From the definition of f', we conclude that f' and $\mid$ commute. Below, we prove that f' commutes with $\bowtie$.

Theorem 1 *f', $\bowtie$ commute.*

Proof: We prove the result for unary f; the general case is similar. Proof is by structural induction.

Base:
$$\begin{aligned} & f'(\langle x\rangle \bowtie \langle y\rangle) \\ = \ & \{\langle x\rangle \bowtie \langle y\rangle = \langle x\rangle \mid \langle y\rangle\} \\ & f'(\langle x\rangle \mid \langle y\rangle) \\ = \ & \{\text{definition of } f'\} \\ & f'\langle x\rangle \mid f'\langle y\rangle \\ = \ & \{f'\langle x\rangle, f'\langle y\rangle = \langle f\ x\rangle, \langle f\ y\rangle. \text{ These are singleton lists}\} \\ & f'\langle x\rangle \bowtie f'\langle y\rangle \end{aligned}$$

Induction:
$$\begin{aligned} & f'((p \mid q) \bowtie (u \mid v)) \\ = \ & \{\ \mid\ ,\ \bowtie \text{ in the argument commute}\} \end{aligned}$$

$$
\begin{array}{ll}
 & f'((p \bowtie u) \mid (q \bowtie v)) \\
= & \{f', \mid \text{ commute}\} \\
 & f'(p \bowtie u) \mid f'(q \bowtie v) \\
= & \{\text{induction}\} \\
 & ((f'\ p) \bowtie (f'\ u)) \mid ((f'\ q) \bowtie (f'\ v)) \\
= & \{\mid, \bowtie \text{ commute}\} \\
 & ((f'\ p) \mid (f'\ q)) \bowtie ((f'\ u) \mid (f'\ v)) \\
= & \{f', \mid \text{ commute}\} \\
 & (f'(p \mid q)) \bowtie (f'(u \mid v))
\end{array}
$$

□

Theorem 2 *For a scalar function f, $f' = f$.*

Proof: Proof by structural induction is straightforward. □

Theorem 3 *If f, g commute then so do f', g'.*

Proof: By structural induction. □

The following results about commutativity can be derived from Theorems 1,2,3. In the following, m, n are natural numbers.

C1. For any f and $m > n$, $f^m, |^n$ commute, and $f^m, \bowtie^n$ commute.
C2. For $m \neq n$, $|^m, |^n$ commute, and $\bowtie^m, \bowtie^n$ commute.
C3. For all m, n, $|^m, \bowtie^n$ commute.
C4. For any scalar function f, $f, |^m$ commute, and $f, \bowtie^n$ commute.

C1 follows by applying induction on Theorems 1 and 3 (and the fact that f', $\mid$ commute). C2 follows from C1; C3 from C1, Law L3 and Theorem 3; C4 from C1 and Theorem 2.

8.5.2 Deconstruction

In this section we show that any powerlist that can be written as $p \ |^m \ q$ for some p, q can also be written as $u \bowtie^m v$ for some u, v and vice versa; this is analogous to Law L1, for dual deconstruction. Analogous to Law L2, we show that such deconstructions are unique.

Theorem 4 *(dual deconstruction): For any p, q and $m \geq 0$, if $p \ |^m \ q$ is defined then there exist u, v such that*

$$u \bowtie^m v = p \ |^m \ q$$

Conversely, for any u, v and $m \geq 0$, if $u \bowtie^m v$ is defined then there exist some p, q such that

$$p \ |^m \ q = u \bowtie^m v$$

□

We do not prove this theorem; its proof is similar to the theorem given below.

Theorem 5 *(unique deconstruction): Let* $\otimes$ *be* $\mid$ *or* $\bowtie$ *. For any natural number* m*,*

$$(p \otimes^m q = u \otimes^m v) \;\equiv\; (p = u \;\wedge\; q = v)$$

Proof: Proof is by induction on m.

$m = 0$: The result follows from Law L2.
$m = n + 1$: Assume that $\otimes = \mid$. The proof is similar for $\otimes = \bowtie$. We prove the result by structural induction on p.

Base: $p = \langle a \rangle$, $q = \langle b \rangle$, $u = \langle c \rangle$, $v = \langle d \rangle$
$\langle a \rangle \mid^{n+1} \langle b \rangle = \langle c \rangle \mid^{n+1} \langle d \rangle$
$\equiv$ {definition of $\mid^{n+1}$}
$\langle a \mid^n b \rangle = \langle c \mid^n d \rangle$
$\equiv$ {unique deconstruction using Law L2}
$a \mid^n b = c \mid^n d$
$\equiv$ {induction on n}
$(a = c) \;\wedge\; (b = d)$
$\equiv$ {Law L2}
$(\langle a \rangle = \langle c \rangle) \;\wedge\; (\langle b \rangle = \langle d \rangle)$

Induction: $p = p_0 \mid p_1$, $q = q_0 \mid q_1$, $u = u_0 \mid u_1$, $v = v_0 \mid v_1$
$(p_0 \mid p_1) \;\mid^{n+1}\; (q_0 \mid q_1) = (u_0 \mid u_1) \;\mid^{n+1}\; (v_0 \mid v_1)$
$\equiv$ {definition of $\mid^{n+1}$}
$(p_0 \mid^{n+1} q_0) \mid (p_1 \mid^{n+1} q_1) = (u_0 \mid^{n+1} v_0) \mid (u_1 \mid^{n+1} v_1)$
$\equiv$ {unique deconstruction using Law L2}
$(p_0 \mid^{n+1} q_0) = (u_0 \mid^{n+1} v_0) \;\wedge\; (p_1 \mid^{n+1} q_1) = (u_1 \mid^{n+1} v_1)$
$\equiv$ {induction on the length of p_0, q_0, p_1, q_1}
$(p_0 = u_0) \;\wedge\; (q_0 = v_0) \;\wedge\; (p_1 = u_1) \;\wedge\; (q_1 = v_1)$
$\equiv$ {Law L2}
$(p_0 \mid p_1) = (u_0 \mid u_1) \;\wedge\; (q_0 \mid q_1) = (v_0 \mid v_1)$

Theorems 4 and 5 allow a richer variety of pattern matching in function definitions, as we did for matrix transposition. We may employ $\mid^m, \bowtie^n$ for any natural m, n to construct a pattern over which a function can be defined.

8.5.3 Embedding Arrays in Hypercubes

An n-dimensional *hypercube* is a graph of 2^n nodes, $n \geq 0$, where each node has a unique n-bit label. Two nodes are *neighbors*, i.e., there is an edge between them, exactly when their labels differ in a single bit. Therefore, every node has n neighbors. We may represent a n-dimensional hypercube as a powerlist of depth n; each level, except the innermost, consists of two powerlists. The operators $\mid^m, \bowtie^n$ for natural m, n can be used to access the nodes in any one (or any combination of) dimensions.

We conclude with an example that shows how higher dimensional structures, such as hypercubes, are easily handled in our theory. Given an array of size $2^{m_0} \times 2^{m_1} \times \ldots 2^{m_d}$, we claim that its elements can be placed at the nodes of a hypercube (of dimension $m_0 + m_1 + .. + m_d$) such that two "adjacent" data items in the array are placed at neighboring nodes in the hypercube. Here, two data items of the array are *adjacent* if their indices differ in exactly one dimension, and by 1 modulo N, where N is the size of that dimension. (This is called "wrap around" adjacency.)

The following embedding algorithm is described in [31, Section 3.1.2]; it works as follows. If the array has only one dimension with 2^m elements, then we create a gray code sequence, $G\ m$ (see Section 8.4.3). Abbreviate $G\ m$ by g. We place the i^{th} item of the array at the node with label g_i. Adjacent items, at positions i and $i+1$ (+ is taken modulo $2^m - 1$), are placed at nodes g_i and g_{i+1} which differ in exactly one bit, by the construction.

This idea can be generalized to higher dimensional arrays as follows. Construct gray code sequences for each dimension independently; store the item with index $(i_0, i_1, \ldots, i_d)$ at the node $(g_{i_0}; g_{i_1}; \ldots; g_{i_d})$ where ";" denotes the concatenations of the bit strings. By definition, adjacent items differ by 1 in exactly one dimension, k. Then, their gray code indices are identical in all dimensions except k and they differ in exactly one bit in dimension k.

We describe a function, em, that embeds an array in a hypercube. Given an array of size $2^{m_0} \times 2^{m_1} \times .. 2^{m_d}$ it permutes its elements to an array $\underbrace{2 \times 2 \times \ldots \times 2}_{m}$, where $m = m_0 + .. + m_d$, and the permutation preserves array adjacency as described. The algorithm is inspired by the gray code function of Section 8.4.3. In the following, S matches only with a scalar and P with a powerlist.

$$
\begin{aligned}
&em\langle S\rangle = \langle S\rangle \\
&em\langle P\rangle = em\ P \\
&em(u \mid v) = \langle em\ u\rangle \mid \langle em\ (rev\ v)\rangle
\end{aligned}
$$

The first line is the rule for embedding a single item in 0-dimensional hypercube. The next line, simply, says that an array having length 1 in a dimension can be embedded by ignoring that dimension. The last line says that a non-singleton array can be embedded by embedding the left half of dimension 0 and the reverse of the right half in the two component hypercubes of a larger hypercube.

8.6 Remarks

Related Work

Applying uniform operations on aggregates of data have proved to be extremely powerful in APL [20]; see [3] and [5] for algebras of such operators. One of the earliest attempts at representing data parallel algorithms is in [39]. In their words, "an algorithm... performs a sequence of basic operations on pairs of

data that are successively $2^{(k-1)}$, $2^{(k-2)}$, .., $2^0 = 1$ locations apart". An algorithm operating on 2^N pieces of data is described as a sequence of N parallel steps of the above form where the k^{th} step, $0 < k \leq N$, applies in parallel a binary operation, OPER, on pairs of data that are $2^{(N-k)}$ apart. They show that this paradigm can be used to describe a large number of known parallel algorithms, and any such algorithm can be efficiently implemented on the Cube Connected Cycle connection structure. Their style of programming was imperative. It is not easy to apply algebraic manipulations to such programs. Their programming paradigm fits in well within our notation. Mou and Hudak[37] and Mou[38] propose a functional notation to describe divide and conquer-type parallel algorithms. Their notation is a vast improvement over Preparata and Vuillemin's in that changing from an imperative style to a functional style of programming allows more succinct expressions and the possibility of algebraic manipulations; the effectiveness of this programming style on a scientific problem may be seen in [50]. They have constructs similar to tie and zip, though they allow unbalanced decompositions of lists. An effective method of programming with vectors has been proposed in [7, 8]. He proposes a small set of "vector-scan" instructions that may be used as primitives in describing parallel algorithms. Unlike our method he is able to control the division of the list and the number of iterations depending on the values of the data items, a necessary ingredient in many scientific problems. Jones and Sheeran[21] have developed a relational algebra for describing circuit components. A circuit component is viewed as a relation and the operators for combining relations are given appropriate interpretations in the circuit domain. Kapur and Subramaniam[22] have implemented the powerlist notation for the purpose of automatic theorem proving. They have proved many of the algorithms in this paper using an inductive theorem prover, called RRL (Rewrite Rule Laboratory), that is based on equality reasoning and rewrite rules. They are now extending their theorem prover so that the similarity constraints on the powerlist constructors do not have to be stated explicitly.

One of the fundamental problems with the powerlist notation is to devise compilation strategies for mapping programs (written in the powerlist notation) to specific architectures. The architecture that is the closest conceptually is the hypercube. Kornerup[28] has developed certain strategies whereby each parallel step in a program is mapped to a constant number of local operations and communications at a hypercube node.

Combinational circuit verification is an area in which the powerlist notation may be fruitfully employed. Adams[1] has proved the correctness of adder circuits using this notation. A ripple-carry adder is typically easy to describe and prove, whereas a carry-lookahead adder is much more difficult. Adams has described both circuits in our notation and proved their equivalence in a remarkably concise fashion. He obtains a succinct description of the carry-lookahead circuit by employing the prefix-sum function (See Section 4.7).

Powerlists of Arbitrary Length

The lengths of the powerlists have been restricted to be of the form 2^n, $n \geq 0$, because we could then develop a simple theory. For handling arbitrary length lists, Steele[45] suggests padding enough "dummy" elements to a list to make its length a power of 2. This scheme has the advantage that we still retain the simple algebraic laws of powerlist. Another approach is based on the observation that any positive integer is either 1 or $2 \times m$ or $2 \times m + 1$, for some positive integer m; therefore, we deconstruct a non-singleton list of odd length into two lists p, q and an element e, where e is either the first or the middle or the last element. For instance, the following function, rev, reverses a list.

$$
\begin{aligned}
&rev\ \langle x \rangle = \langle x \rangle \\
&rev\ (p \mid q) = (rev\ q) \mid (rev\ p) \\
&rev\ (p \mid e \mid q) = (rev\ q \mid e \mid rev\ p)
\end{aligned}
$$

The last line of this definition applies to a non-singleton list of odd length; the list is deconstructed into two lists p, q of equal length and e, the middle element. (We have abused the notation, applying $\mid$ to three arguments). Similarly, the function $\langle\forall$ for prefix sum may be defined by

$$
\begin{aligned}
&\langle\forall\langle x \rangle = \langle x \rangle \\
&\langle\forall(p \bowtie q) = (t^* \oplus p) \bowtie t \\
&\langle\forall(e \bowtie p \bowtie q) = e \bowtie (e \oplus (t^* \oplus p)) \bowtie (e \oplus t) \\
&\qquad \text{where } t = \langle\forall(p \oplus q)
\end{aligned}
$$

In this definition, the singleton list and lists of even length are treated as before. A list of odd length is deconstructed into e, p, q, where e is the first element of the argument list and $p \bowtie q$ constitutes the remaining portion of the list. For this case, the prefix sum is obtained by appending the element e to the list obtained by applying $e\oplus$ to each element of $\langle\forall(p \bowtie q)$; we have used the convention that $(e \oplus L)$ is the list obtained by applying $e\oplus$ to each element of list L.

The Interplay between Sequential and Parallel Computations.

The notation proposed in this paper addresses only a small aspect of parallel computing. Powerlists have proved to be highly successful in expressing computations that are independent of the specific data values; such is the case, for instance, in the Fast Fourier Transform, Batcher merge and prefix sum. Typically, however, parallel and sequential computations are interleaved. While Fast Fourier Transform and Batcher merge represent highly parallel computations, binary search is inherently sequential (there are other parallel search strategies). Gaussian elimination represents a mixture; the computation consists of a sequence of pivoting steps where each step can be applied in parallel. Thus

parallel computations may have to be performed in a certain sequence and the sequence may depend on the data values during a computation. More general methods, as in [7], are then required.

The powerlist notation can be integrated into a language that supports sequential computation. In particular, this notation blends well with ML [35] and LISP[34, 46]. A mixture of linear lists and powerlists can exploit the various combinations of sequential and parallel computing. A powerlist consisting of linear lists as components admits of parallel processing in which each component is processed sequentially. A linear list whose elements are powerlists suggests a sequential computation where each step can be applied in parallel. Powerlists of powerlists allow multidimensional parallel computations, whereas a linear list of linear lists may represent a hierarchy of sequential computations.

Bibliography

[1] Will Adams. Verifying adder circuits using powerlists. Technical Report TR 94-02, Dept. of Computer Science, Univ. of Texas at Austin, Austin, Texas 78712, Mar 1994.

[2] Alfred V. Aho and Jeffrey D Ullman. *Foundations of Computer Science*. W. H. Freeman, 1995.

[3] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, Aug 1978. Turing Award Lecture (1977).

[4] Kenneth Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, Reston, VA, 1968. AFIPS Press.

[5] R. S. Bird. Lectures on constructive functional programming. In Manfred Broy, editor, *Constructive Methods in Computing Science*, NATO ASI Series F: Computer and Systems Sciences, pages 151–216. Springer-Verlag, 1989.

[6] Richard Bird. *Introduction to Functional Programming using Haskell*. International Series in Computer Science, C.A.R. Hoare and Richard Bird, Series Editors. Prentice-Hall International, 1998.

[7] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[8] Guy E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-93-129, Carnegie-Mellon Univ., School of Computer Science, Apr 1993.

[9] M. Burrows and D. J. Wheeler. A Block-sorting Lossles Data Compression Algorithm. Technical Report 124, Digital, SRC Research Report, May 1994.

[10] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[11] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[12] J. M. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19(90):297–301, 1965.

[13] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. Inform. Theory*, 22(6):644–654, 1976.

[14] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972. Turing Award lecture.

[15] Edsger W. Dijkstra. Pruning the search tree, EWD 1255. `http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1255.PDF`, January 1997. circulated privately.

[16] G.-J.Nam, K.A.Sakallah, and R.Rutenbar. Satisfiability based FPGA routing. *Proceedings of the International Conference on VLSI Design*, January 1999.

[17] F. Gray. Pulse code communication. U.S. Patent 2,632,058, Mar 1953.

[18] Haskell 98: A non-strict, purely functional language. Available at `http://haskell.org/onlinereport`, 1999.

[19] Paul Hudak, Jon Peterson, and Joseph Fasel. A Gentle Introduction to Haskell, Version 98. Available at `http://www.haskell.org/tutorial/`, 2000.

[20] K. Iverson. *A Programming Language*. John Wiley and Sons, 1962.

[21] Geraint Jones and Mary Sheeran. Circuit design in Ruby. In Jørgen Staunstrup, editor, *Formal Methods for VLSI Design*. North-Holland, 1990.

[22] D. Kapur and M. Subramaniam. Automated reasoning about parallel algorithms using powerlists. Manuscipt in preparation, 1994.

[23] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier and the MIT Press, 1990.

[24] Henry Kautz and Bart Selman. Planning as satisfiability. *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 92)*, 1992.

[25] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 1997.

[26] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 1998.

[27] Donald E. Knuth. The Complexity of Songs. *Communications of the ACM*, 27(4):344–348, Apr 1984.

[28] Jacob Kornerup. Mapping a functional notation for parallel programs onto hypercubes. *Information Processing Letters*, 53:153–158, 1995.

[29] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the Association for Computing Machinery*, 27(4):831–838, 1980.

[30] Tracy Larrabee. *Efficient generation of test patterns using boolean satisfiability*. PhD thesis, Stanford University, 1990.

[31] F. Thompson Leighton. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann Publishers, San Mateo, California, 1992.

[32] Philip M. Lewis, Arthur Bernstein, and Michael Kifer. *Databases and Transaction Processing*. Addison-Wesley, 2002.

[33] J. Marques-Silva and K.A.Sakallah. Boolean satisfiability in electronic design automation. *Proceedings of the ACM/IEEE Design Automation Conference*, June 2000.

[34] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual*. MIT Press, 1962.

[35] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[36] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT solver. In *Proceedings of the 39th Design Automation Conference*, June 2001.

[37] Z. G. Mou and P. Hudak. An algebraic model for divide-and-conquer algorithms and its parallelism. *The Journal of Supercomputing*, 2(3):257–278, Nov 1988.

[38] Z.G. Mou. Divacon: A parallel language for scientific computing based on divide-and-conquer. In *Proc. 3rd Symp. on the Frontiers of Massively Parallel Computation*, pages 451–461, Oct 1991.

[39] Franco P. Preparata and Jean Vuillemin. The cube-connected cycles: A versatile network for parallel computation. *Communications of the ACM*, 24(5):300–309, May 1981.

[40] Ham Richards. Overhead foils for lectures on Haskell. Available at `http://www.cs.utexas.edu/users/ham/CS378Slides.pdf`, 2002.

[41] R.L. Rivest, A. Shamir, and L. Adelman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, Feb 1978.

[42] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 1948.

[43] Claude E. Shannon. Prediction and entropy of printed English. *The Bell System Technical Journal*, 30:50–64, 1950.

[44] Simon Singh. *The Code Book*. Doubleday, 1999.

[45] Guy L. Steele Jr. Personal communication, 1993.

[46] Guy L. Steele Jr. and Daniel Hillis. Connection Machine Lisp: Fine-grained parallel symbolic processing. In *Proc. 1986 ACM Conference on Lisp and Functional Programming*, pages 279–297, Cambridge, Mass., Aug 1986. ACM SIGPLAN/SIGACT/SIGART.

[47] The home page for the mChaff and zChaff sat solvers. `http://www.ee.princeton.edu/~chaff/software.php`, 2002.

[48] Simon Thompson. *The Craft of Functional Programming*. Addison-Wesley Longman, Harlow, Essex, 1996.

[49] David Turner. An overview of Miranda. *ACM SIGPLAN Notices*, 21:156–166, Dec 1986.

[50] X. Wang and Z.G. Mou. A divide-and-conquer method of solving tridiagonal systems on hypercube massively parallel computers. In *proc. of the 3rd IEEE symposium on parallel and distributed processing*, pages 810–817, Dallas, Tx., Dec 1991.

[51] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, IT-24(5):530–536, Sept. 1978.