Part 0
Why Use Logic?
Why Prove Programs Correct?

A story

We have just finished writing a large program (3000 lines). Among other things, the program computes as intermediate results the quotient q and remainder r arising from dividing a non-negative integer x by a positive integer y. For example, with $x = 7$ and $y = 2$, the program calculates $q = 3$ (since $7 \div 2 = 3$) and $r = 1$ (since the remainder when 7 is divided by 2 is 1).

Our program appears below, with dots "..." representing the parts of the program that precede and follow the remainder-quotient calculation. The calculation is performed as given because the program will sometimes be executed on a micro-computer that has no integer division, and portability must be maintained at all costs! The remainder-quotient calculation actually seems quite simple; since $\div$ cannot be used, we have elected to repeatedly subtract divisor y from a copy of x, keeping track of how many subtractions are made, until another subtraction would yield a negative integer.

```
...
r := x; q := 0;
while r ≥ y do
  begin r := r − y; q := q + 1 end;
...
```

We're ready to debug the program. With respect to the remainder-quotient calculation, we're smart enough to realize that the divisor should initially be greater than 0 and that upon its termination the variables should satisfy the formula

$$x = y*q + r,$$

so we add some output statements to check the calculations:

```
...
write('dividend x =', x, 'divisor y =', y);
r := x; q := 0;
while r > y do
  begin r := r - y; q := q + 1 end;
write('y*q + r =', y*q + r);
...
```

Unfortunately, we get voluminous output because the program segment occurs in a loop, so our first test run is wasted. We try to be more selective about what we print. Actually, we need to know values only when an error is detected. Having heard of a new feature just inserted into the compiler, we decide to try it. If a Boolean expression appears within braces { and } at a point in the program, then, whenever "flow of control" reaches that point during execution, it is checked: if false, a message and a dump of the program variables are printed; if true, execution continues normally. These Boolean expressions are called *assertions*, since in effect we are asserting that they should be true when flow of control reaches them. The systems people encourage leaving assertions in the program, because they help document it.

Protests about inefficiency during production runs are swept aside by the statement that there is a switch in the compiler to turn off assertion checking. Also, after some thought, we decide it may be better to always check assertions —detection of an error during production would be well worth the extra cost.

So we add assertions to the program:

```
...
{y > 0}
r := x; q := 0;
(1) while r > y do
      begin r := r - y; q := q + 1 end;
{x = y*q + r}
...
```

Testing now results in far less output, and we make progress. Assertion checking detects an error during a test run because y is 0 just before a remainder-quotient calculation, and it takes only four hours to find the error in the calculation of y and fix it.

But then we spend a day tracking down an error for which we received no nice false-assertion message. We finally determine that the remainder-quotient calculation resulted in

$$x = 6,\ y = 3,\ q = 1,\ r = 3.$$

Sure enough, both assertions in (1) are true with these values; the problem is that the remainder should be less than the divisor, and it isn't. We determine that the loop condition should be $r \geq y$ instead of $r > y$. If only the result assertion were strong enough —if only we had used the assertion $x = y*q + r$ **and** $r < y$— we would have saved a day of work! Why didn't we think of it?

We fix the error and insert the stronger assertion:

```
...
{y > 0}
r := x; q := 0;
while r ≥ y do
   begin r := r − y; q := q + 1 end;
{x = y*q + r and r < y}
...
```

Things go fine for a while, but one day we get incomprehensible output. It turns out that the quotient-remainder algorithm resulted in a negative remainder $r = -2$. But the remainder shouldn't be negative! And we find out that r was negative because initially x was -2. Ahhh, another error in calculating the input to the quotient-remainder algorithm —x isn't *supposed* to be negative! But we could have caught the error earlier and saved two days searching, in fact we *should* have caught it earlier; all we had to do was make the initial and final assertions for the program segment strong enough. Once more we fix an error and strengthen an assertion:

```
...
{0 ≤ x and 0 < y}
r := x; q := 0;
while r ≥ y do
   begin r := r − y; q := q + 1 end;
{x = y*q + r and 0 ≤ r < y}
...
```

It sure would be nice to be able to invent the right assertions to use in a less *ad hoc* fashion. Why can't we think of them? Does it have to be a trial-and-error process? Part of our problem here was carelessness in specifying what the program segment was to do —we should have written

the initial assertion ($0 \leqslant x$ **and** $0 < y$) and the final assertion ($x = y*q + r$ **and** $0 \leqslant r < y$) *before* writing the program segment, for they form the definition of quotient and remainder.

But what about the error we made in the condition of the while loop? Could we have prevented that from the beginning? Is there is a way to prove, just from the program and assertions, that the assertions are true when flow of control reaches them? Let's see what we can do.

Just before the loop it seems that part of our result,

(2) $x = y*q + r$

holds, since $x = r$ and $q = 0$. And from the assignments in the loop body we conclude that if (2) is true before execution of the loop body then it is true after its execution, so it will be true just before and after *every* iteration of the loop. Let's insert it as an assertion in the obvious places, and let's also make all assertions as *strong* as possible:

```
. . .
{0 ⩽ x and 0 < y}
r := x; q := 0;
{0 ⩽ r and 0 < y and x = y*q + r}
while r ⩾ y do
  begin {0 ⩽ r and 0 < y ⩽ r and x = y*q + r}
        r := r − y; q := q + 1
        {0 ⩽ r and 0 < y and x = y*q + r}
end;
{0 ⩽ r < y and x = y*q + r}
. . .
```

Now, how can we easily determine a correct loop condition, or, given the condition, how can we prove it is correct? When the loop terminates the condition is false. Upon termination we want $r < y$, so that the complement, $r \geqslant y$ must be the correct loop condition. How easy that was!

It seems that if we knew how to make all assertions as strong as possible and if we learned how to reason carefully about assertions and programs, then we wouldn't make so many mistakes, we would *know* our program was correct, and we wouldn't need to debug programs at all! Hence, the days spent running test cases, looking through output and searching for errors could be spent in other ways.

Discussion

The story suggests that assertions, or simply Boolean expressions, are really needed in programming. But it is not enough to know how to write Boolean expressions; one needs to know how to *reason* with them: to simplify them, to prove that one follows from another, to prove that one is not true in some state, and so forth. And, later on, we will see that it is necessary to use a kind of assertion that is not part of the usual Boolean expression language of Pascal, PL/I or FORTRAN, the "quantified" assertion.

Knowing how to reason about assertions is one thing; knowing how to reason about *programs* is another. In the past 10 years, computer science has come a long way in the study of proving programs correct. We are reaching the point where the subject can be taught to undergraduates, or to anyone with some training in programming and the will to become more proficient. More importantly, the study of program correctness proofs has led to the discovery and elucidation of methods for *developing* programs. Basically, one attempts to develop a program and its proof hand-in-hand, with the proof ideas leading the way! If the methods are practiced with care, they can lead to programs that are free of errors, that take much less time to develop and debug, and that are much more easily understood (by those who have studied the subject).

Above, I mentioned that programs could be free of errors and, in a way, I implied that debugging would be unnecessary. This point needs some clarification. Even though we can become more proficient in programming, we will still make errors, even if only of a syntactic nature (typos). We are only human. Hence, some testing will always be necessary. But it should not be called debugging, for the word debugging implies the existence of bugs, which are terribly difficult to eliminate. No matter how many flies we swat, there will always be more. A disciplined method of programming should give more confidence than that! We should run test cases not to look for bugs, but to increase our confidence in a program we are quite sure is correct; finding an error should be the exception rather than the rule.

With this motivation, let us turn to our first subject, the study of logic.

Part I
Propositions and Predicates

Chapter 1 defines the syntax of propositions —Boolean expressions using only Boolean variables— and shows how to evaluate them. Chapter 2 gives rules for manipulating propositions, which is often done in order to find simpler but equivalent ones. This chapter is important for further work on programming, and should be studied carefully.

Chapter 3 introduces a *natural deduction system* for proving theorems about propositions, which is supposed to mimic in some sense the way we "naturally" argue. Such systems are used in research on mechanical verification of proofs of program correctness, and one should become familiar with them. But the material is not needed to understand the rest of the book and may be skipped entirely.

Chapter 4 extends propositions to include variables of types besides Boolean and introduces quantification. A *predicate calculus* is given, in which one can express and manipulate the assertions we make about program variables. "Bound" and "free" variables are introduced and the notion of textual substitution is studied. This material is necessary for further reading.

Chapter 5 concerns arrays. Thinking of an array as a function from subscript values to array element values, instead of as a collection of independent variables, leads to some neat notation and rules for dealing with arrays. The first two sections of this chapter should be read, but the third may be skipped on first reading.

Finally, chapter 6 discusses briefly the use of assertions in programs, thus motivating the next two parts of the book.

Chapter 1
Propositions

We want to be able to describe sets of states of program variables and to write and manipulate clear, unambiguous assertions about program variables. We begin by considering only variables (and expressions) of type *Boolean*: from the operational point of view, each variable contains one of the values T and F, which represent our notions of "truth" and "falsity", respectively. The word *Boolean* comes from the name of a 19th century English mathematician, George Boole, who initiated the algebraic study of truth values.

Like many logicians, we will use the word *proposition* for the kind of Boolean or logical expression to be defined and discussed in this chapter.

Propositions are similar to arithmetic expressions. There are operands, which represent the values T or F (instead of integers), and operators (e.g. **and**, **or** instead of *, +), and parentheses are used to aid in determining order of evaluation. The problem will not be in defining and evaluating propositions, but in learning how to express assertions written in English as propositions and to reason with those propositions.

1.1 Fully Parenthesized Propositions

Propositions are formed according to the following rules (the operators will be defined subsequently). As can be seen, parentheses are required around each proposition that includes an operation. This restriction, which will be weakened later on, allows us to dispense momentarily with problems of precedence of operators.

1. T and F are propositions.
2. An identifier is a proposition. (An identifier is a sequence of one or more digits and letters, the first of which is a letter.)

3. If b is a proposition, then so is $(\neg b)$.

4. If b and c are propositions, then so are $(b \wedge c)$, $(b \vee c)$, $(b \Rightarrow c)$, and $(b = c)$.

This syntax may be easier to understand in the form of a BNF grammar (Appendix 1 gives a short introduction to BNF):

(1.1.1) <proposition> ::= T | F | <identifier>
| ($\neg$ <proposition>)
| (<proposition> $\wedge$ <proposition>)
| (<proposition> $\vee$ <proposition>)
| (<proposition> $\Rightarrow$ <proposition>)
| (<proposition> = <proposition>)

Example. The following are propositions (separated by commas):

F, $(\neg T)$, $(b \vee xyz)$, $((\neg b) \wedge (c \Rightarrow d))$,
$((abc1 = id) \wedge (\neg d))$ □

Example. The following are not propositions:

$F\ F$, $(b \vee (c)$, $(b) \wedge)$, $a + b$ □

As seen in the above syntax, five operators are defined over values of type *Boolean*:

negation:	(**not** b), or $(\neg b)$
conjunction:	(b **and** c), or $(b \wedge c)$
disjunction:	(b **or** c), or $(b \vee c)$
implication:	(b **imp** c), or $(b \Rightarrow c)$
equality:	(b **equals** c), or $(b = c)$

Two different notations have been given for each operator, a name and a mathematical symbol. The name indicates how to pronounce it, and its use also makes typing easier when a typewriter does not have the corresponding mathematical symbol.

The following terminology is used. $(b \wedge c)$ is called a *conjunction*; its operands b and c called *conjuncts*. $(b \vee c)$ is called a *disjunction*; its operands b and c are called *disjuncts*. $(b \Rightarrow c)$ is called an *implication*; its *antecedent* is b and its *consequent* is c.

1.2 Evaluation of Constant Propositions

Thus far we have given a *syntax* for propositions; we have defined the set of well-formed propositions. We now give a semantics (meaning) by showing how to evaluate them.

We begin by defining evaluation of *constant propositions* —propositions that contain only constants as operands— and we do this in three cases based on the structure of a proposition e: for e with no operators, for e with one operator, and for e with more than one operator.

(1.2.1) **Case 1.** The value of proposition T is T; the value of F is F.

(1.2.2) **Case 2.** The values of $(\neg b)$, $(b \wedge c)$, $(b \vee c)$, $(b \Rightarrow c)$ and $(b = c)$, where b and c are each one of the constants T and F, are given by the following table (called a *truth table*). Each row of the table contains possible values for the operands b and c and, for these values, shows the value of each of the five operations. For example, from the last row we see that the value of $(\neg T)$ is F and that the values of $(T \wedge T)$, $(T \vee T)$, $(T \Rightarrow T)$ and $(T = T)$ are all T.

(1.2.3)

b	c	$(\neg b)$	$(b \wedge c)$	$(b \vee c)$	$(b \Rightarrow c)$	$(b = c)$
F	F	T	F	F	T	T
F	T	T	F	T	T	F
T	F	F	F	T	F	F
T	T	F	T	T	T	T

(1.2.4) **Case 3.** The value of a constant proposition with more than one operator is found by repeatedly applying (1.2.2) to a subproposition of the constant proposition and replacing the subproposition by its value, until the proposition is reduced to T or F.

We give an example of evaluation of a proposition:

$$\begin{aligned} &((T \wedge T) \Rightarrow F) \\ &= (T \Rightarrow F) \\ &= F \end{aligned}$$

Remark: The description of the operations in terms of a truth table, which lists *all* possible operand combinations and their values, can be given only because the set of possible values is finite. For example, no such table could be given for operations on integers. □

The names of the operations correspond fairly closely to their meanings in English. For example, "not true" usually means "false", and "not

false" "true". But note that operation **or** denotes "inclusive or" and not "exclusive or". That is, $(T \vee T)$ is T, while the "exclusive or" of T and T is false.

Also, there is no causality implied by operation **imp**. The sentence "If it rains, the picnic is cancelled" can be written in propositional form as (*rain* ⇒ *no picnic*). From the English sentence we infer that the lack of rain means there will be a picnic, but *no* such inference can be made from the proposition (*rain* ⇒ *no picnic*).

1.3 Evaluation of Propositions in a State

A proposition like $((\neg c) \vee d)$ can appear in a program in several places, for example in an assignment statement $b := ((\neg c) \vee d)$ and in an **if**-statement **if** $((\neg c) \vee d)$ **then** $\cdots$. When the statement in which the proposition appears is to be executed, the proposition is evaluated in the current machine "state" to produce either T or F. To define this evaluation requires a careful explanation of the notion of "state".

A state associates identifiers with values. For example, in state s (say), identifier c could be associated with value F and identifier d with T. In terms of a computer memory, when the computer is in state s, locations named c and d contain the values F and T, respectively. In another state, the associations could be (c, T) and (d, F). The crucial point here is that a state consists of a set of pairs (identifier, value) in which all the identifiers are distinct, i.e. the state is a function:

(1.3.1) **Definition**. A *state* s is a function from a set of identifiers to the set of values T and F. □

Example. Let state s be the function defined by the set $[(a, T), (bc, F), (yl, T)]$. Then $s(a)$ denotes the value determined by applying state (function) s to identifier a: $s(a) = T$. Similarly, $s(bc) = F$ and $s(yl) = T$. □

(1.3.2) **Definition**. Proposition e is *well-defined in state* s if each identifier in e is associated with either T or F in state s. □

In state $s = [(b, T), (c, F)]$, proposition $(b \vee c)$ is well-defined while proposition $(b \vee d)$ is not.

Let us now extend the notation $s(identifier)$ to define the value of a proposition in a state. For any state s and proposition e, $s(e)$ will denote the value resulting from evaluating e in state s. Since an identifier b is also a proposition, we will be careful to make sure that $s(b)$ will still denote the value of b in state s.

(1.3.3) **Definition.** Let proposition e be well-defined in state s. Then $s(e)$, the value of e in state s, is the value obtained by replacing all occurrences of identifiers b in e by their values $s(b)$ and evaluating the resulting constant proposition according to the rules given in the previous section 1.2. □

Example. $s(((\neg b) \vee c))$ is evaluated in state $s = [(b, T), (c, F)]$:

$$\begin{aligned} & s(((\neg b) \vee c)) \\ & \quad = ((\neg T) \vee F) \quad (b \text{ has been replaced by } T,\ c \text{ by } F) \\ & \quad = (F \vee F) \\ & \quad = F \qquad \square \end{aligned}$$

1.4 Precedence Rules for Operators

The previous sections dealt with a restricted form of propositions, so that evaluation of propositions could be explained without having to deal with the precedence of operators. We now relax this restriction.

Parentheses can be omitted or included at will around any proposition. For example, the proposition $((b \vee c) \Rightarrow d)$ can be written as $b \vee c \Rightarrow d$. In this case, additional rules define the order of evaluation of subpropositions. These rules, which are similar to those for arithmetic expressions are:

1. Sequences of the same operator are evaluated from left to right, e.g. $b \wedge c \wedge d$ is equivalent to $((b \wedge c) \wedge d)$.

2. The order of evaluation of different, adjacent operators is given by the list: **not** (has highest precedence and binds tightest), **and, or, imp, equals.**

It is usually better to make liberal use of parentheses in order to make the order of evaluation clear, and we will usually do so.

Examples

$\neg b = b \wedge c$	is equivalent to	$(\neg b) = (b \wedge c)$	
$b \vee \neg c \Rightarrow d$	is equivalent to	$(b \vee (\neg c)) \Rightarrow d$	
$b \Rightarrow c \Rightarrow d \wedge e$	is equivalent to	$(b \Rightarrow c) \Rightarrow (d \wedge e)$	□

The following BNF grammar defines the syntax of propositions, giving enough structure so that precedences can be deduced from it. (The non-terminal <identifier> has been left undefined and has its usual meaning).

1. <proposition> ::= <imp-expr>
2. | <proposition> ≡ <imp-expr>
3. <imp-expr> ::= <expr>
4. | <imp-expr> ⇒ <expr>
5. <expr> ::= <term>
6. | <expr> ∨ <term>
7. <term> ::= <factor>
8. | <term> ∧ <factor>
9. <factor> ::= ¬ <factor>
10. | (<proposition>)
11. | *T*
12. | *F*
13. | <identifier>

We now define $s(e)$, the value of proposition e in state s, recursively, based on the structure of e given by the grammar. That is, for each rule of the grammar, we show how to evaluate e if it has the form given by that rule. For example, rule 6 indicates that for an <expr> of the form <expr> ∨ <term>, its value is the value found by applying operation or to the values $s($<expr>$)$ and $s($<term>$)$ of its operands <expr> and <term>. The values of the five operations ≡, ⇒, ∨, ∧ and ¬ used in rules 2, 4, 6, 8 and 9 are given by truth table (1.2.3).

1. $s($<proposition>$)$ = $s($<imp-expr>$)$
2. $s($<proposition>$)$ = $s($<proposition>$)$ ≡ $s($<imp-expr>$)$
3. $s($<imp-expr>$)$ = $s($<expr>$)$
4. $s($<imp-expr>$)$ = $s($<imp-expr>$)$ ⇒ $s($<expr>$)$
5. $s($<expr>$)$ = $s($<term>$)$
6. $s($<expr>$)$ = $s($<expr>$)$ ∨ $s($<term>$)$
7. $s($<term>$)$ = $s($<factor>$)$
8. $s($<term>$)$ = $s($<term>$)$ ∧ $s($<factor>$)$
9. $s($<factor>$)$ = ¬ $s($<factor>$)$
10. $s($<factor>$)$ = $s($<proposition>$)$
11. $s($<factor>$)$ = *T*
12. $s($<factor>$)$ = *F*
13. $s($<factor>$)$ = $s($<identifier>$)$ (the value of <identifier> in s)

An example of evaluation using a truth table

Let us compute values of the proposition $(b \Rightarrow c) \equiv (\neg b \vee c)$ for all possible operand values using a truth table. In the table below, each row gives possible values for b and c and the corresponding values of $\neg b$, $\neg b \vee c$, $b \Rightarrow c$ and the final proposition. This truth table shows how one builds a truth table for a proposition, by beginning with the values of the

identifiers, then showing the values of the smallest subpropositions, then the next smallest, and building up to the complete proposition.

As can be seen, the values of $\neg b \vee c$ and $b \Rightarrow c$ are the same in each state, and hence the propositions are equivalent and can be used interchangeably. In fact, one often finds $b \Rightarrow c$ *defined* as $\neg b \vee c$. Similarly, $b = c$ is often defined as an abbreviation for $(b \Rightarrow c) \wedge (c \Rightarrow b)$ (see exercise 2i).

b	c	$\neg b$	$\neg b \vee c$	$b \Rightarrow c$	$(b \Rightarrow c) = (\neg b \vee c)$
F	F	T	T	T	T
F	T	T	T	T	T
T	F	F	F	F	T
T	T	F	T	T	T

1.5 Tautologies

A *Tautology* is a proposition that is true in every state in which it is well-defined. For example, proposition T is a tautology and F is not. The proposition $b \vee \neg b$ is a tautology, as can be seen by evaluating it with $b = T$ and $b = F$:

$$T \vee \neg T = T \vee F = T$$
$$F \vee \neg F = F \vee T = T$$

or, in truth-table form:

b	$\neg b$	$b \vee \neg b$
T	F	T
F	T	T

The basic way to show that a proposition is a tautology is to show that its evaluation yields T in every possible state. Unfortunately, each extra identifier in a proposition doubles the number of combinations of values for identifiers —for a proposition with i distinct identifiers there are 2^i cases! Hence, the work involved can become tedious and time consuming. To illustrate this, (1.5.1) contains the truth table for proposition $(b \wedge c \wedge d) \Rightarrow (d \Rightarrow b)$, which has three distinct identifiers. By taking some shortcuts, the work can be reduced. For example, a glance at truth table (1.2.3) indicates that operation **imp** is true whenever its antecedent is false, so that its consequent need only be evaluated if its antecedent is true. In example (1.5.1) there is only one state in which the antecedent $b \wedge c \wedge d$ is true —the state in which b, c and d are true— and hence we need only the top line of truth table (1.5.1).

(1.5.1)

b	c	d	$b \wedge c \wedge d$	$d \Rightarrow b$	$(b \wedge c \wedge d) \Rightarrow (d \Rightarrow b)$
T	T	T	T	T	T
T	T	F	F	T	T
T	F	T	F	T	T
T	F	F	F	T	T
F	T	T	F	F	T
F	T	F	F	T	T
F	F	T	F	F	T
F	F	F	F	T	T

Using such informal reasoning helps reduce the number of states in which the proposition must be evaluated. Nevertheless, the more distinct identifiers a proposition has the more states to inspect, and evaluation soon becomes infeasible. Later chapters investigate other methods for proving that a proposition is a tautology.

Disproving a conjecture

Sometimes we conjecture that a proposition e is a tautology, but are unable to develop a proof of it, so we decide to try to disprove it. What does it take to disprove such a conjecture?

It may be possible to prove the converse —i.e. that $\neg e$ is a tautology— but the chances are slim. If we had reason to believe a conjecture, it is unlikely that its converse is true. Much more likely is that it is true in most states but false in one or two, and to disprove it we need only find one such state:

> *To prove a conjecture, it is necessary to prove that it is true in all cases; to disprove a conjecture, it is sufficient to find a single case where it is false.*

1.6 Propositions as Sets of States

A proposition *represents*, or describes, the set of states in which it is true. Conversely, for any set of states containing only identifiers associated with T or F we can derive a proposition that represents that state set. Thus, the empty set, the set containing no states, is represented by proposition F because F is true in no state. The set of all states is represented by proposition T because T is true in all states. The following example illustrates how one can derive a proposition that represents a given set of states. The resulting proposition contains only the operators **and**, **or** and **not**.

Example. The set of two states $\{(b,T),(c,T),(d,T)\}$ and $\{(b,F),(c,T),(d,F)\}$, is represented by the proposition

$$(b \wedge c \wedge d) \vee (\neg b \wedge c \wedge \neg d) \quad \square$$

The connection between a proposition and the set of states it represents is so strong that we often *identify* the two concepts. Thus, instead of writing "the set of states in which $b \vee \neg c$ is true" we may write "the states in $b \vee \neg c$". Though it is a sloppy use of English, it is at times convenient.

In connection with this discussion, the following terminology is introduced. Proposition b is *weaker* than c if $c \Rightarrow b$. Correspondingly, c is said to be *stronger* than b. A stronger proposition makes more restrictions on the combinations of values its identifiers can be associated with, a weaker proposition makes fewer. In terms of sets of states, b is as weak as c if it is "less restrictive": if b's set of states includes at least c's states, and possibly more. The weakest proposition is T (or any tautology), because it represents the set of all states; the strongest is F, because it represents the set of no states.

1.7 Transforming English to Propositional Form

At this point, we translate a few sentences into propositional form. Consider the sentence "If it rains, the picnic is cancelled." Let identifier r stand for the proposition "it rains" and let identifier pc represent "the picnic is cancelled". Then the sentence can be written as $r \Rightarrow pc$.

As shown by this example, the technique is to represent "atomic parts" of a sentence —how these are chosen is up to the translator— by identifiers and to describe their relationship using Boolean operators. Here are some more examples, using identifiers r, pc, wet, and s defined as follows:

it rains: r
picnic is cancelled: pc
be wet: wet
stay at home: s

1. If it rains but I stay at home, I won't be wet: $(r \wedge s) \Rightarrow \neg wet$
2. I'll be wet if it rains: $r \Rightarrow wet$
3. If it rains and the picnic is not cancelled or I don't stay home, I'll be wet: Either $((r \wedge \neg pc) \vee \neg s) \Rightarrow wet$ or $(r \wedge (\neg pc \vee \neg s)) \Rightarrow wet$. The English is ambiguous; the latter proposition is probably the desired one.

4. Whether or not the picnic is cancelled, I'm staying home if it rains: $(pc \vee \neg pc) \wedge r \Rightarrow s$. This reduces to $r \Rightarrow s$.

5. Either it doesn't rain or I'm staying home: $\neg r \vee s$.

Exercises for Chapter 1

1. Each line contains a proposition and two states *s1* and *s2*. Evaluate the proposition in both states.

proposition	state *s1*				state *s2*			
	m	n	p	q	m	n	p	q
(a) $\neg(m \vee n)$	T	F	T	T	F	T	T	T
(b) $\neg m \vee n$	T	F	T	T	F	T	T	T
(c) $\neg(m \wedge n)$	T	F	T	T	F	T	T	T
(d) $\neg m \wedge n$	T	F	T	T	F		T	
(e) $(m \vee n) \Rightarrow p$	T	F	T	T	T	T	F	T
(f) $m \vee (n \Rightarrow p)$	T	F	T	T	T	T	F	T
(g) $(m = n) \wedge (p = q)$	F	F	T	F	T	F	T	F
(h) $m = (n \wedge (p = q))$	F	F	T	F	T	F	T	F
(i) $m = (n \wedge p = q)$	F	F	T	F	T	F	T	F
(j) $(m = n) \wedge (p \Rightarrow q)$	F	T	F	T	T	T	F	F
(k) $(m = n \wedge p) \Rightarrow q$	F	T	F	T	T	T	F	F
(l) $(m \Rightarrow n) \Rightarrow (p \Rightarrow q)$	F	F	F	F	T	T	T	T
(m) $(m \Rightarrow (n \Rightarrow p)) \Rightarrow q$	F	F	F	F	T	T	T	T

2. Write truth tables to show the values of the following propositions in all states:

(a) $b \vee c \vee d$
(b) $b \wedge c \wedge d$
(c) $b \wedge (c \vee d)$
(d) $b \vee (c \wedge d)$
(e) $\neg b \Rightarrow (b \vee c)$
(f) $\neg b = (b \vee c)$
(g) $(\neg b = c) \vee b$
(h) $(b \vee c) \wedge (b \Rightarrow c) \wedge (c \Rightarrow b)$
(i) $(b = c) = (b \Rightarrow c) \wedge (c \Rightarrow b)$

3. Translate the following sentences into propositional form.

(a) $x < y$ or $x = y$.
(b) Either $x < y$, $x = y$, or $x > y$.
(c) If $x > y$ and $y > z$, then $v = w$.
(d) The following are all true: $x < y$, $y < z$ and $v = w$.
(e) At most one of the following is true: $x < y$, $y < z$ and $v = w$.
(f) None of the following are true: $x < y$, $y < z$ and $v = w$.
(g) The following are not all true at the same time: $x < y$, $y < z$ and $v = w$.
(h) When $x < y$, then $y < z$; when $x \geq y$, then $v = w$.
(i) When $x < y$ then $y < z$ means that $v = w$, but if $x \geq y$ then $y < z$ doesn't hold; however, if $v = w$ then $x < y$.
(j) If execution of program P is begun with $x < y$, then execution terminates with $y = 2^x$.
(k) Execution of program P begun with $x < 0$ will not terminate.

4. Below are some English sentences. Introduce identifiers to represent the simple ones (e.g. "it's raining cats and dogs.") and then translate the sentences into propositions.

(a) Whether or not it's raining, I'm going swimming.
(b) If it's raining I'm not going swimming.
(c) It's raining cats and dogs.
(d) It's raining cats or dogs.
(e) If it rains cats and dogs I'll eat my hat, but I won't go swimming.
(f) If it rains cats and dogs while I am swimming I'll eat my hat.

Chapter 2
Reasoning using Equivalence Transformations

Evaluating propositions is rarely our main task. More often we wish to manipulate them in some manner in order to derive "equivalent" but simpler ones (easier to read and understand). Two propositions (or, in general, expressions) are equivalent if they have the same value in every state. For example, since $a+(c-a)=c$ is always true for integer variables a and c, the two integer expressions $a+(c-a)$ and c are equivalent, and $a+(c-a)=c$ is called an equivalence.

This chapter defines equivalence of propositions in terms of the evaluation model of chapter 1. A list of useful equivalences is given, together with two rules for generating others. The idea of a "calculus" is discussed, and the rules are put in the form of a formal calculus for "reasoning" about propositions.

These rules form the basis for much of the manipulations we do with propositions and are very important for later work on developing programs. The chapter should be studied carefully.

2.1 The Laws of Equivalence

For propositions, we define equivalence in terms of operation **equals** and the notion of a tautology as follows:

(2.1.1) **Definition**. Propositions *E1* and *E2* are *equivalent iff* $E1=E2$ is a tautology. In this case, $E1=E2$ is an *equivalence*. □

Thus, an equivalence is an equality that is a tautology.

Below, we give a list of equivalences; these are the basic equivalences from which all others will be derived, so we call them the *laws of equivalence*. Actually, they are "schemas": the identifiers *E1*, *E2* and *E3*

within them are parameters, and one arrives at a particular equivalence by substituting particular propositions for them. For example, substituting $x \vee y$ for $E1$ and z for $E2$ in the first law of Commutativity, $(E1 \wedge E2) = (E2 \wedge E1)$, yields the equivalence

$$((x \vee y) \wedge z) = (z \wedge (x \vee y))$$

Remark: Parentheses are inserted where necessary when performing a substitution so that the order of evaluation remains consistent with the original proposition. For example, the result of substituting $x \vee y$ for b in $b \wedge z$ is $(x \vee y) \wedge z$, and not $x \vee y \wedge z$, which is equivalent to $x \vee (y \wedge z)$. □

1. **Commutative Laws** (These allow us to reorder the operands of **and**, **or** and **equality**):
   $(E1 \wedge E2) = (E2 \wedge E1)$
   $(E1 \vee E2) = (E2 \vee E1)$
   $(E1 = E2) = (E2 = E1)$

2. **Associative Laws** (These allow us to dispense with parentheses when dealing with sequences of **and** and sequences of **or**):
   $E1 \wedge (E2 \wedge E3) = (E1 \wedge E2) \wedge E3$ (so write both as $E1 \wedge E2 \wedge E3$)
   $E1 \vee (E2 \vee E3) = (E1 \vee E2) \vee E3$

3. **Distributive Laws** (These are useful in factoring a proposition, in the same way that we rewrite $2*(3+4)$ as $(2*3)+(2*4)$):
   $E1 \vee (E2 \wedge E3) = (E1 \vee E2) \wedge (E1 \vee E3)$
   $E1 \wedge (E2 \vee E3) = (E1 \wedge E2) \vee (E1 \wedge E3)$

4. **De Morgan's Laws** (After Augustus De Morgan, a 19th century English mathematician who, along with Boole, laid much of the foundations for mathematical logic):
   $\neg(E1 \wedge E2) = \neg E1 \vee \neg E2$
   $\neg(E1 \vee E2) = \neg E1 \wedge \neg E2$

5. **Law of Negation**: $\neg(\neg E1) = E1$

6. **Law of the Excluded Middle**: $E1 \vee \neg E1 = T$

7. **Law of Contradiction**: $E1 \wedge \neg E1 = F$

8. **Law of Implication**: $E1 \Rightarrow E2 = \neg E1 \vee E2$

9. **Law of Equality**: $(E1 = E2) = (E1 \Rightarrow E2) \wedge (E2 \Rightarrow E1)$

10. **Laws of or-simplification**:
$$E1 \vee E1 = E1$$
$$E1 \vee T = T$$
$$E1 \vee F = E1$$
$$E1 \vee (E1 \wedge E2) = E1$$

11. **Laws of and-simplification**:
$$E1 \wedge E1 = E1$$
$$E1 \wedge T = E1$$
$$E1 \wedge F = F$$
$$E1 \wedge (E1 \vee E2) = E1$$

12. **Law of Identity**: $E1 = E1$

Don't be alarmed at the number of laws. Most of them you have used many times, perhaps unknowingly, and this list will only serve to make you more aware of them. Study the laws carefully, for they are used over and over again in manipulating propositions. Do some of the exercises at the end of this section until the use of these laws becomes second nature. Knowing the laws by name makes discussions of their use easier.

The law of the Excluded Middle deserves some comment. It means that either b or $\neg b$ must be true in any state; there can be no middle ground. Some don't believe this law, at least in all its generality. In fact, here is a counterexample to it, in English. Consider the sentence

> This sentence is false.

which we might consider as the meaning of an identifier b. Is it true or false? It can't be true, because it says it is false; it can't be false, because then it would be true! The sentence is neither true nor false, and hence violates the law of the Excluded Middle. The paradox arises because of the self-referential aspect of the sentence —it indicates something about itself, as do all paradoxes. [Here is another paradox to ponder: a barber in a small town cuts the hair of every person in town except for those who cut their own. Who cuts the barber's hair?] In our formal system, there will be no way to introduce such self-referential treatment, and the law of the Excluded Middle holds. But this means we cannot express *all* our thoughts and arguments in the formal system.

Finally, the laws of Equality and Implication deserve special mention. Together, they define **equality** and **Imp** in terms of other operators: $b = c$ can always be replaced by $(b \Rightarrow c) \wedge (c \Rightarrow b)$ and $\neg b \Rightarrow c$ by $b \vee c$. This reinforces what we said about the two operations in chapter 1.

Proving that the logical laws are equivalences

We have stated, without proof, that laws 1-12 are equivalences. One way to prove this is to build truth tables and note that the laws are true in all states. For example, the first of De Morgan's laws, $\neg(E1 \wedge E2) = \neg E1 \vee \neg E2$, has the following truth table:

$E1$	$E2$	$E1 \wedge E2$	$\neg(E1 \wedge E2)$	$\neg E1$	$\neg E2$	$\neg E1 \vee \neg E2$	$\neg(E1 \wedge E2) = \neg E1 \vee \neg E2$
F	F	F	T	T	T	T	T
F	T	F	T	T	F	T	T
T	F	F	T	F	T	T	T
T	T	T	F	F	F	F	T

Clearly, the law is true in all states (in which it is well-defined), so that it is a tautology.

Exercise 1 concerns proving all the laws to be equivalences.

2.2 The Rules of Substitution and Transitivity

Thus far, we have just discussed some basic equivalences. We now turn to ways of generating other equivalences, without having to check their truth tables. One rule we all use in transforming expressions, usually without explicit mention, is the rule of "substitution of equals for equals". Here is an example of the use of this rule. Since $a+(c-a)=c$, we can substitute for expression $a+(c-a)$ in $(a+(c-a))*d$ to conclude that $(a+(c-a))*d=c*d$; we simply replace $a+(c-a)$ in $(a+(c-a))*d$ by the simpler, but equivalent, expression c.

The rule of substitution is:

(2.2.1) **Rule of Substitution.** Let $e1=e2$ be an equivalence and $E(p)$ be a proposition, written as a function of one of its identifiers p. Then $E(e1)=E(e2)$ and $E(e2)=E(e1)$ are also equivalences. □

Here is an example of the use of the rule of Substitution. The law of Implication indicates that $(b \Rightarrow c)=(\neg b \vee c)$ is an equivalence. Consider the proposition $E(p)=d \vee p$. With

$$e1 = b \Rightarrow c \quad \text{and}$$
$$e2 = \neg b \vee c$$

we have

$$E(e1) = d \vee (b \Rightarrow c)$$
$$E(e2) = d \vee (\neg b \vee c)$$

so that $(d \vee (b \Rightarrow c) = d \vee (\neg b \vee c)$ is an equivalence.

In using the rule of Substitution, we often use the following form. The proposition that we conclude is an equivalence is written on one line. The initial proposition appears to the left of the equality sign and the one that results from the substitution appears to the right, followed by the name of the law $e1 = e2$ used in the application:

$$d \vee (b \Rightarrow c) = d \vee (\neg b \vee c) \quad \text{(Implication)}$$

We need one more rule for generating equivalences:

(2.2.2) **Rule of Transitivity**. If $e1 = e2$ and $e2 = e3$ are equivalences, then so is $e1 = e3$ (and hence $e1$ is equivalent to $e3$). □

Example. We show that $(b \Rightarrow c) = (\neg c \Rightarrow \neg b)$ is an equivalence (an explanation of the format follows):

$$\begin{aligned} & b \Rightarrow c \\ &= \neg b \vee c && \text{(Implication)} \\ &= c \vee \neg b && \text{(Commutativity)} \\ &= \neg\neg c \vee \neg b && \text{(Negation)} \\ &= \neg c \Rightarrow \neg b && \text{(Implication)} \end{aligned}$$

This is read as follows. First, lines 1 and 2 indicate that $b \Rightarrow c$ is equivalent to $\neg b \vee c$, by virtue of the rule of Substitution and the law of Implication. Secondly, lines 2 and 3 indicate that $(\neg b \vee c)$ is equivalent to $c \vee \neg b$, by virtue of the rule of Substitution and the law of Commutativity. We also conclude, using the rule of Transitivity, that the first proposition, $b \Rightarrow c$, is equivalent to the third, $c \vee \neg b$. Continuing in this fashion, each pair of lines gives an equivalence and the reasons why the equivalence holds. We finally conclude that the first proposition, $b \Rightarrow c$, is equivalent to the last, $\neg c \Rightarrow \neg b$. □

Example. We show that the law of Contradiction can be proved from the others. The portion of each proposition to be replaced in each step is underlined in order to make it easier to identify the substitution.

$$\begin{aligned} \underline{\neg(b \wedge \neg b)} &= \neg b \vee \underline{\neg \neg b} && \text{(De Morgan's Law)} \\ &= \underline{\neg b \vee b} && \text{(Negation)} \\ &= \underline{b \vee \neg b} && \text{(Commutativity)} \\ &= T && \text{(Excluded Middle)} \quad \square \end{aligned}$$

Generally speaking, such fine detail is unnecessary. The laws of Commutativity and Associativity are often used without explanation, and the application of several steps can appear on one line. For example:

$$
\begin{aligned}
&(b \wedge (b \Rightarrow c)) \Rightarrow c \\
&\quad = \neg(b \wedge (\neg b \vee c)) \vee c && \text{(Implication, 2 times)} \\
&\quad = \neg b \vee \neg(\neg b \vee c) \vee c && \text{(De Morgan)} \\
&\quad = T && \text{(Excluded Middle)}
\end{aligned}
$$

Transforming an implication

Suppose we want to prove that

(2.2.3) $E1 \wedge E2 \wedge E3 \Rightarrow E$

is an equivalence. The proposition is transformed as follows:

$$
\begin{aligned}
&(E1 \wedge E2 \wedge E3) \Rightarrow E \\
&\quad = \neg(E1 \wedge E2 \wedge E3) \vee E && \text{(Implication)} \\
&\quad = \neg E1 \vee \neg E2 \vee \neg E3 \vee E && \text{(De Morgan)}
\end{aligned}
$$

The final proposition is true in any state in which at least one of $\neg E1$, $\neg E2$, $\neg E3$ and E is true. Hence, to prove that (2.2.3) is a tautology we need only prove that in any state in which three of them are false the fourth is true. And we can choose which three to assume false, based on their form, in order to develop the simplest proof.

With an argument similar to the one just given, we can see that the five statements

$$
\begin{aligned}
&E1 \wedge E2 \wedge E3 \Rightarrow E \\
&E1 \wedge E2 \wedge \neg E \Rightarrow \neg E3 \\
&E1 \wedge \neg E \wedge E3 \Rightarrow \neg E2 \\
&\neg E \wedge E2 \wedge E3 \Rightarrow \neg E1 \\
&\neg E1 \vee \neg E2 \vee \neg E3 \vee E
\end{aligned} \qquad (2.2.4)
$$

are equivalent and we can choose which to work with. When given a proposition like (2.2.3), eliminating implication completely in favor of disjunctions like (2.2.4) can be helpful. Likewise, when formulating a problem, put it in the form of a disjunction right from the beginning.

Example. Prove that

$$(\neg(b \Rightarrow c) \wedge \neg(\neg b \Rightarrow (c \vee d))) \Rightarrow (\neg c \Rightarrow d)$$

is a tautology. Eliminate the main implication and use De Morgan's law:

$$\neg\neg(b \Rightarrow c) \vee \neg\neg(\neg b \Rightarrow (c \vee d)) \vee (\neg c \Rightarrow d)$$

Now simplify using Negation and eliminate the other implications:

$$(\neg b \vee c) \vee (b \vee c \vee d) \vee (c \vee d)$$

Use the laws of Associativity, Commutativity and **or**-simplification to arrive at

$$b \vee \neg b \vee c \vee d$$

which is true because of the laws of the Excluded Middle, $b \vee \neg b = T$, and **or**-simplification. This problem, which at first looked quite difficult, became simple when the implications were eliminated.

2.3 A Formal System of Axioms and Inference Rules

A *calculus*, according to Webster's Third International Dictionary, is a method or process of reasoning by computation of symbols. In section 2.2 we presented a calculus, for by performing some symbol manipulation according to rules of Substitution and Transitivity we can reason with propositions. For obvious reasons, the system presented here is called a *propositional calculus*.

We are careful to say *a* propositional calculus, and not *the* propositional calculus. With slight changes in the rules we can have a different calculus. Or we can invent a completely different set of rules and a completely different calculus, which is better suited for other purposes.

We want to emphasize the nature of this calculus as a formal system for manipulating propositions. To do this, let us put aside momentarily the notions of state and evaluation and see whether equivalences, which we will call *theorems*, can be discussed without them. First, define the propositions that arise directly from laws 1-12 to be *theorems*. They are also called *axioms* (and the laws 1-12 are *axiom schemas*), because their *theorem*hood is taken at face value, without proof.

(2.3.1) **Axioms**. Any proposition that arises by substituting propositions for *E1*, *E2* and *E3* in one of the Laws 1-12 is called a *theorem*. □

Next, define the propositions that arise by using the rules of Substitution and Transitivity and an already-derived *theorem* to be a *theorem*. In this context, the rules are often called *inference rules*, for they can be used to *infer* that a proposition is a *theorem*. An inference rule is often written in the form

$$\frac{E_1, \cdots, E_n}{E} \quad \text{and} \quad \frac{E_1, E_2, \cdots, E_n}{E, E_9}$$

where the E_i and E stand for arbitrary propositions. The inference rule has the following meaning. If propositions $E_1, \cdots, E_n$ are *theorems*, then so is proposition E (and E_0 in the second case). Written in this form, the rules of Substitution and Transitivity are

(2.3.2) **Rule of Substitution**: $$\frac{e1 = e2}{E(e1) = E(e2),\ E(e2) = E(e1)}$$

(2.3.3) **Rule of Transitivity**: $$\frac{e1 = e2,\ e2 = e3}{e1 = e3}$$

A *theorem* of the formal system, then, is either an axiom (according to (2.3.1) or a proposition that is derived from one of the inference rules (2.3.2) and (2.3.3).

Note carefully that this is a totally different system for dealing with propositions, which has been defined without regard to the notions of states and evaluation. The syntax of propositions is the same, but what we do with propositions is entirely different. Of course, there is a relation between the formal system and the system of evaluation given in the previous chapter. Exercises 9 and 10 call for proof of the following relationship: for any tautology e in the sense of chapter 1, $e = T$ is a *theorem*, and vice versa.

Exercises for Chapter 2

1. Verify that laws 1-12 are equivalences by building truth tables for them.

2. Prove the law of Identity, $e = e$, using the rules of Substitution and Transitivity and the laws 1-11.

3. Prove that $\neg T = F$ is an equivalence, using the rules of Substitution and Transitivity and the laws 1-12.

4. Prove that $\neg F = T$ is an equivalence, using the rules of Substitution and Transitivity and the laws 1-12.

5. Each column below consists of a sequence of propositions, each of which (except the first) is equivalent to its predecessor. The equivalence can be shown by one application of the rule of Substitution and one of the laws 1-12 or the results of exercises 3-4. Identify the law (as is done for the first two cases).

(a) $(x \wedge y) \vee (z \wedge \neg z)$	(a) $\neg(\neg b \wedge (\neg b \Rightarrow z)) \vee z$
(b) $(x \wedge y) \vee F$ Contradiction	(b) $(\neg b \wedge (\neg b \Rightarrow z)) \Rightarrow z)$
(c) $x \wedge y$ or-simplification	(c) $(\neg b \wedge (\neg \neg b \vee z)) \Rightarrow z$
(d) $(x \wedge y) \vee F$	(d) $(\neg b \wedge (\neg \neg b \vee \neg \neg z)) \Rightarrow z$
(e) $(x \wedge y) \vee (F \wedge z)$	(e) $(\neg b \wedge \neg(\neg b \wedge \neg z)) \Rightarrow z$
(f) $(x \wedge y) \vee (F \wedge z)$	(f) $(\neg b \wedge \neg(\neg b \wedge \neg z)) \Rightarrow z$
(g) $(x \wedge y) \vee ((x \wedge \neg x) \wedge z)$	(g) $\neg(b \vee (\neg b \wedge \neg z)) \Rightarrow z$

(h) $(x \wedge y) \vee (x \wedge (\neg x \wedge z))$
(i) $x \wedge (y \vee (\neg x \wedge z))$
(j) $x \wedge (y \vee \neg x) \wedge (y \vee z)$
(k) $x \wedge (\neg x \vee y) \wedge (z \vee y)$
(l) $x \wedge (\neg x \vee \neg \neg y) \wedge (z \vee y)$
(m) $x \wedge \neg (x \wedge \neg y) \wedge (z \vee y)$

(h) $\neg((b \vee \neg b) \wedge (b \vee \neg z)) \Rightarrow z$
(i) $\neg(T \wedge (b \vee \neg z)) \Rightarrow z$
(j) $\neg(b \vee \neg z) \Rightarrow z$
(k) $\neg \neg (b \vee \neg z) \vee z$
(l) $(b \vee \neg z) \vee z)$
(m) $b \vee (\neg z \vee z)$

6. Each proposition below can be simplified to one of the six propositions F, T, x, y, $x \wedge y$, and $x \vee y$. Simplify them, using the rules of Substitution and Transitivity and the laws 1-12.

(a) $x \vee (y \vee x) \vee \neg y$
(b) $(x \vee y) \wedge (x \vee \neg y)$
(c) $x \vee y \vee \neg x$
(d) $(x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y)$
(e) $(x \wedge y) \vee (x \wedge \neg y) \vee (\neg x \wedge y) \vee (\neg x \wedge \neg y)$
(f) $(\neg x \wedge y) \vee x$

(g) $\neg x \Rightarrow (x \wedge y)$
(h) $T \Rightarrow (\neg x \Rightarrow x)$
(i) $x \Rightarrow (y \Rightarrow (x \wedge y))$
(j) $\neg x \Rightarrow (\neg x \Rightarrow (\neg x \wedge y))$
(k) $\neg y \Rightarrow y$
(l) $\neg y \Rightarrow \neg y$

7. Show that any proposition e can be transformed into an equivalent proposition in *disjunctive normal form* —i.e. one that has the form

$$e_0 \vee \cdots \vee e_n \text{ where each } e_i \text{ has the form } g_0 \wedge \cdots \wedge g_m$$

Each g_j is an identifier id, a unary operator $\neg id$, T or F. Furthermore, the identifiers in each e_i are distinct.

8. Show that any proposition e can be transformed into an equivalent proposition in *conjunctive normal form* —i.e. one that has the form

$$e_0 \wedge \cdots \wedge e_n \text{ where each } e_i \text{ has the form } g_0 \vee \cdots \vee g_m$$

Each g_j is an identifier id, a unary operator $\neg id$, T or F. Furthermore, the identifiers in each e_i are distinct.

9. Prove that any *theorem* generated using laws 1-12 and the rules of Substitution and Transitivity is a tautology, by proving that laws 1-12 are tautologies (see exercise 1) and showing that the two rules can generate only tautologies.

10. Prove that if e is a tautology, then $e = T$ can be proved to be an equivalence using only the laws 1-12 and the rules of Substitution and Transitivity. Hint: use exercise 8.

Chapter 3
A Natural Deduction System

This chapter introduces another formal system of axioms and inference rules for deducing proofs that propositions are tautologies. It is called a "natural deduction system" because it is meant to mimic the patterns of reasoning that we "naturally" use in making arguments in English.

This material is not used in later parts of the book, and can be skipped. The equivalence transformation system discussed in chapter 2 serves more than adequately in developing correct programs later on. One could go further and say that the equivalence transformation system is more suited to our needs, although, this may be a matter of taste.

Nevertheless, study of this chapter is worthwhile for several reasons. The formal system presented here is minimal: there are *no* axioms and a minimal number of inference rules. Thus, one can see what it takes to start with a bare-bones system and build up enough theorems to the point where further theorems are not cumbersome to prove. The equivalence transformation system, on the other hand, provided as axioms all the useful basic equivalences. Secondly, such systems are being used more and more in mechanical verification systems, and the computer science student should be familiar with them. (A natural deduction system is also used in the popular game WFF'N PROOF.) Finally, it is useful to see and compare two totally different formal systems for dealing with propositions.

3.1 Introduction to Deductive Proofs

Consider the problem of proving that a conclusion follows from certain premises. For example, we might want to prove that $p \wedge (r \vee q)$ follows from $p \wedge q$ —i.e. $p \wedge (r \vee q)$ is true in every state in which $p \wedge q$ is. This problem can be written in the following form:

(3.1.1) premise: $p \wedge q$
conclusion: $p \wedge (r \vee q)$

In English, we might argue as follows.

(3.1.2) *Proof of (3.1.1):* Since $p \wedge q$ is true (in state s), so is p, and so is q. One property of **or** is that, for any r, $r \vee q$ is true if q is, so $r \vee q$ is true. Finally, since p and $r \vee q$ are both true, the properties of **and** allow us to conclude that $p \wedge (r \vee q)$ is true in s also.

In order to get at the essence of such proofs, in order to determine just what is involved in such arguments, we are going to strip away the verbiage from the proof and present simply the bare details. Admittedly, the proofs will look (at first) complicated and detailed. But once we have worked with the proof method for a while, we will be able to return to informal proofs in English with much better facility. We will also be able to give some guidelines for developing proofs (section 3.5).

The bare details of proof (3.1.2) are, in order: a statement of the theorem, the sequence of propositions initially assumed to be true, and the sequence of propositions that are true based on previous propositions and various rules of inference.

These bare details are presented in (3.1.3). The first line states the theorem to be proved: "**From** $p \wedge q$ **infer** $p \wedge (r \vee q)$". The second line gives the premise (if there were more premises, they would be given on successive lines). Each of the succeeding lines gives a proposition that one can infer, based on the truth of the propositions in the previous lines and an inference rule. The last line contains the conclusion.

(3.1.3)

From $p \wedge q$ **infer** $p \wedge (r \vee q)$

1	$p \wedge q$	premise
2	p	property of **and**, 1
3	q	property of **and**, 1
4	$r \vee q$	property of **or**, 3
5	$p \wedge (r \vee q)$	property of **and**, 2, 4

To the right of each proposition appears an explanation of how the proposition's "truth" is derived. For example, line 4 of the proof indicates

that $r \vee q$ is true because of a property of **or** —that $r \vee q$ is true if q is— and because q appears on the preceding line 3. Note that parentheses are introduced freely in order to maintain priority of operators. We shall continue to do this without formal description.

In this formal system, a theorem to be proved has the form

From $e_1, \cdots, e_n$ **infer** e.

In terms of evaluation of propositions, such a theorem is interpreted as: if $e_1, \ldots, e_n$ are true in a state, then e is true in that state also. If n is 0, meaning that there are no premises, then it can be interpreted as: e is true in all states, i.e. e is a tautology. In this case we write it as

Infer e.

Finally, a proposition on a line of a proof can be interpreted to mean that it is true in any state in which the propositions on previous lines are true.

As mentioned earlier, our natural deduction system has no axioms. The properties of operators used above are captured in the inference rules, which we begin to introduce and explain in the next section. (Inference rules were first introduced in section 2.3; review that material if necessary.) The inference rules for the natural deduction system are collected in Figure 3.3.1 at the end of section 3.3.

3.2 Inference Rules

There are ten inference rules in the natural deduction system. Ten is a rather large number, and we can work with that many only if they are organized so that they are easy to remember. In this system, there are two inference rules for each of the five operators **not**, **and**, **or**, **imp** and **equals**. One of the rules allows the introduction of the operator in a new proposition; the other allows its elimination. Hence there are five rules of introduction and five rules of elimination. The rules for introducing and eliminating **and** are called $\wedge$-I and $\wedge$-E, respectively, and similarly for the other operators.

Inference rules $\wedge$-I, $\wedge$-E and $\vee$-I

Let us begin by giving three rules: $\wedge$-I, $\wedge$-E and $\vee$-I.

(3.2.1) $\wedge$-I: $\dfrac{E_1, \cdots, E_n}{E_1 \wedge \cdots \wedge E_n}$

(3.2.2) $\wedge$-E: $\dfrac{E_1 \wedge \cdots \wedge E_n}{E_i}$

(3.2.3) $\vee$-I: $\dfrac{E_i}{E_1 \vee \cdots \vee E_n}$

Rule $\wedge$-I indicates that if E_1 and E_2 occur on previous lines of a proof (i.e. are assumed to be true or have been proved to be true), then their conjunction may be written on a line. If we assert "it is raining", and we assert "the sun is shining", then we can conclude "it is raining and the sun is shining". The rule is called "$\wedge$-Introduction", or "$\wedge$-I" for short, because it shows how a conjunction can be introduced.

Rule $\wedge$-E shows how **and** can be eliminated to yield one of its conjuncts. If $E_1 \wedge E_2$ appears on a previous line of a proof (i.e. is assumed to be true or has been proved to be true), then either E_1 or E_2 may be written on the next line. Based on the assumption "it is raining and the sun is shining", we can conclude "it is raining", and we can conclude "the sun is shining".

Remark: There *are* places where it frequently rains while the sun is shining. Ithaca, the home of Cornell University, is one of them. In fact, it sometimes rains when perfectly blue sky seems to be overhead. The weather can also change from a furious blizzard to bright, calm sunshine and then back again, within minutes. When the weather acts so strangely, as it often does, one says that it is *Ithacating*. □

Rule $\vee$-I indicates that if E_1 is on a previous line, then we may write $E_1 \vee E_2$ on a line. If we assert "it is raining", then we can conclude "it is raining or the sun is shining".

Remember, these rules hold for *all* propositions E_1 and E_2. They are really "schemas", and we get an instance of the rule by replacing E_1 and E_2 by particular propositions. For example, since $p \vee q$ and r are propositions, the following is an instance of $\wedge$-I.

$$\frac{p \vee q,\ \neg r}{(p \vee q) \wedge \neg r}$$

Let us redo proof (3.1.3) in (3.2.4) below and indicate the exact inference rule used at each step. The top line states what is to be proved. The line numbered 1 contains the first (and only) premise (pr 1). Each other line has the following property. Let the line have the form

line # | E "name of rule", line #, ..., line #

Then one can form an instance of the named inference rule by writing the propositions on lines line #, ..., line # above a line and proposition E below. That is, the truth of E is inferred by one inference rule from the truth of previous propositions. For example, from line 4 of the proof we see that $q \,/\, r \vee q$ is an instance of rule ∨-I: $(r \vee q)$ is being inferred from q.

(3.2.4)

From $p \wedge q$ infer $p \wedge (r \vee q)$

1	$p \wedge q$	pr 1
2	p	∧-E, 1
3	q	∧-E, 1
4	$r \vee q$	∨-I, 3
5	$p \wedge (r \vee q)$	∧-I, 2, 4

Note how rule ∧-E is used to break a proposition into its constituent parts, while ∧-I and ∨-I are used to build new ones. This is typical of the use of introduction and elimination rules.

Proofs (3.2.5) and (3.2.6) below illustrate that **and** is a commutative operation; if $p \wedge q$ is true then so is $q \wedge p$, and vice versa. This is obvious after our previous study of propositions, but it must be proved in this formal system before it can be used. Note that both proofs are necessary; one cannot derive the second as an instance of the first by replacing p and q in the first by q and p, respectively. In this formal system, a proof holds only for the particular propositions involved. It is not a schema, the way an inference rule is.

(3.2.5)

From $p \wedge q$ infer $q \wedge p$

1	$p \wedge q$	pr 1
2	p	∧-E, 1
3	q	∧-E, 1
4	$q \wedge p$	∧-I, 3, 2

To illustrate the relation between the proof system and English, we give an argument in English for lemma (3.2.5): Suppose $p \wedge q$ is true [line 1]. Then so is p, and so is q [lines 2 and 3]. Therefore, by the definition of **and**, $q \wedge p$ is true [line 4].

(3.2.6)

From $q \wedge p$ infer $p \wedge q$

1	$q \wedge p$	pr 1
2	q	∧-E, 1
3	p	∧-E, 1
4	$p \wedge q$	∧-I, 3, 2

Proof (3.2.6) can be abbreviated by omitting lines containing premises and

using "pr i" to refer to the i^{th} premise later on, as shown in (3.2.7). This abbreviation will occur often. But note that this is only an abbreviation, and we will continue to use the phrase "occurs on a previous line" to include the premises, even though the abbreviation is used.

(3.2.7)

	From $q \wedge p$ infer $p \wedge q$	
1	q	$\wedge$-E, pr 1
2	p	$\wedge$-E, pr 1
3	$p \wedge q$	$\wedge$-I, 2, 1

Inference rule $\vee$-*E*

The inference rule for elimination of **or** is

$$\text{(3.2.8)}\quad \vee\text{-E:}\ \frac{E_1 \vee \cdots \vee E_n, E_1 \Rightarrow E, \cdots, E_n \Rightarrow E}{E}$$

Rule $\vee$-E indicates that if a disjunction appears a previous line, and if $E_i \Rightarrow E$ appears on a previous line for each disjunct E_i, then E may be written on a line of the proof. If we assert "it will rain tomorrow or it will snow tomorrow", and if we assert "rain implies no sun", and if we also assert "snow implies no sun", then we can conclude "there will be no sun tomorrow". From

$$(rain \vee snow), (rain \Rightarrow no\ sun), (snow \Rightarrow no\ sun)$$

we conclude *no sun*.

Here is a simple example.

	From $p \vee (q \wedge r)$, $p \Rightarrow s$, $(q \wedge r) \Rightarrow s$ infer $s \vee p$	
1	$p \vee (q \wedge r)$	pr 1
2	$p \Rightarrow s$	pr 2
3	$(q \wedge r) \Rightarrow s$	pr 3
4	s	$\vee$-E, 1, 2, 3
5	$s \vee p$	$\vee$-I (rule (3.2.3)), 4

Inference rule $\Rightarrow$-*E*

$$\text{(3.2.9)}\quad \Rightarrow\text{-E:}\ \frac{E1 \Rightarrow E2,\ E1}{E2}$$

Rule $\Rightarrow$-E is called *modus ponens*. It allows us to write the consequent of an implication on a line of the proof if its antecedent appears on a previous line. If we assert that $x > 0$ implies that y is even, and if we determine that $x > 0$, then we can conclude that y is even.

We show an example of its use in proof (3.3.10). To show the relation between the formal proof and an English one, we give the proof in English: Suppose $p \wedge q$ and $p \Rightarrow r$ are both true. From $p \wedge q$ we conclude that p is true. Because $p \Rightarrow r$, the truth of p implies the truth of r, and r is true. But if r is true, so is r "ored" with anything; hence $r \vee (q \Rightarrow r)$ is true.

(3.2.10)

From $p \wedge q$, $p \Rightarrow r$ **infer** $r \vee (q \Rightarrow r)$

1	$p \wedge q$	pr 1
2	$p \Rightarrow r$	pr 2
3	p	$\wedge$-E (rule (3.2.2)), 1
4	r	$\Rightarrow$-E, 2, 3
5	$r \vee (q \Rightarrow r)$	$\vee$-I (rule (3.2.3)), 4

To emphasize the use of the abbreviation to refer to premises, we show (3.2.10) in its abbreviated form in (3.2.11).

(3.2.11)

From $p \wedge q$, $p \Rightarrow r$ **infer** $r \vee (q \Rightarrow r)$

1	p	$\wedge$-E, pr 1
2	r	$\Rightarrow$-E, pr 2, 1
3	$r \vee (q \Rightarrow r)$	$\vee$-I, 2

Inference rules $=$-I and $=$-E

(3.2.12) $=$-I: $$\frac{E1 \Rightarrow E2,\ E2 \Rightarrow E1}{E1 = E2}$$

(3.2.13) $=$-E: $$\frac{E1 = E2}{E1 \Rightarrow E2,\ E2 \Rightarrow E1}$$

Rules $=$-I and $=$-E together define equality in terms of implication. The premises of one rule are the conclusions of the other, and vice versa. This is quite similar to how equality is defined in the system of chapter 2. Rule $=$-I is used, then, to introduce an equality $e1 = e2$ based on the previous proof of $e1 \Rightarrow e2$ and $e2 \Rightarrow e1$.

Here is an example of the use of these rules.

From p, $p \Rightarrow (q \Rightarrow r)$, $r \Rightarrow q$ infer $r \equiv q$		
1	$p \Rightarrow (q \Rightarrow r)$	$\equiv$-E, pr 2
2	$q \Rightarrow r$	$\Rightarrow$-E, 1, pr 1
3	$r \equiv q$	$\equiv$-I, pr 3, 2

Exercises for Section 3.2

1. Each of the following theorems can be proven using exactly one basic inference rule (using the abbreviation that premises need not be written on lines; see the text preceding (3.2.7)). Name that inference rule.

(a) **From** a, b **infer** $a \wedge b$
(b) **From** $a \wedge b \wedge (q \vee r)$, a **infer** $q \vee r$
(c) **From** $\neg a$ **infer** $\neg a \vee a$
(d) **From** $c \equiv d$, $d \vee c$ **infer** $d \Rightarrow c$
(e) **From** $b \Rightarrow c$, b **infer** $b \vee \neg b$
(f) **From** $\neg a$, $\neg b$, c **infer** $\neg a \vee c$
(g) **From** $(a \Rightarrow b) \wedge b$, a **infer** $a \Rightarrow b$
(h) **From** $a \vee b \Rightarrow c$, $c \Rightarrow a \vee b$ **infer** $a \vee b \equiv c$
(i) **From** $a \wedge b$, $q \vee r$ **infer** $(a \wedge b) \wedge (q \vee r)$
(j) **From** $p \Rightarrow (q \Rightarrow r)$, p, $q \vee r$ **infer** $q \Rightarrow r$
(k) **From** $c \Rightarrow d$, $d \Rightarrow e$, $d \Rightarrow c$ **infer** $c \equiv d$
(l) **From** $a \vee b$, $a \vee c$, $(a \vee b) \Rightarrow c$ **infer** c
(m) **From** $a \Rightarrow (d \vee c)$, $(d \vee c) \Rightarrow a$ **infer** $a \equiv (d \vee c)$
(n) **From** $(a \vee b) \Rightarrow c$, $(a \vee d) \Rightarrow c$, $(a \vee b) \vee (a \vee d)$ **infer** c
(o) **From** $a \Rightarrow (b \vee c)$, $b \Rightarrow (b \vee c)$, $a \vee b$ **infer** $b \vee c$

2. Here is one proof that p follows from p. Write another proof that uses only one reference to the premise.

From p infer p		
1	p	pr 1
2	p	pr 1

3. Prove the following theorems using the inference rules.

(a) **From** $p \wedge q$, $p \Rightarrow r$ **infer** r
(b) **From** $p \equiv q$, q **infer** p
(c) **From** p, $q \Rightarrow r$, $p \Rightarrow r$ **infer** $p \wedge r$
(d) **From** $b \wedge \neg c$ **infer** $\neg c$
(e) **From** b **infer** $b \vee \neg c$
(f) **From** $b \Rightarrow c \wedge d$, b **infer** d
(g) **From** $p \wedge q$, $p \Rightarrow r$ **infer** r
(h) **From** p, $q \wedge (p \Rightarrow s)$ **infer** $q \wedge s$
(i) **From** $p \equiv q$ **infer** $q \equiv p$
(j) **From** $b \Rightarrow (c \wedge d)$, b **infer** d

4. For each of your proofs of exercise 3, give an English version. (The English versions need not mimic the formal proofs exactly.)

3.3 Proofs and Subproofs

Inference rule ⇒-I

A theorem of the form "**From** $e_1, \cdots, e_n$ **infer** e" is interpreted as: if $e_1, \ldots, e_n$ are true in a state, then so is e. If $e_1, \ldots, e_n$ appear on lines of a proof, which is interpreted to mean that they are assumed or proven true, then we should be able to write e on a line also. Rule ⇒-I, (3.3.1), gives us permission to do so. Its premise need not appear on a previous line of the proof; it can appear elsewhere as a separate proof, which we refer to in substantiating the use of the rule. Unique names should be given to proofs to avoid ambiguous references.

$$\text{(3.3.1)} \quad \Rightarrow\text{-I:} \ \frac{\textbf{From } E_1, \cdots, E_n \textbf{ infer } E}{(E_1 \wedge \cdots \wedge E_n) \Rightarrow E}$$

Proof (3.3.2) uses ⇒-I twice in order to prove that $p \wedge q$ and $q \wedge p$ are equivalent, using lemmas proved in the previous section.

(3.3.2) **Infer** $(p \wedge q) = (q \wedge p)$

1	$(p \wedge q) \Rightarrow (q \wedge p)$	⇒-I, (3.2.5)
2	$(q \wedge p) \Rightarrow (p \wedge q)$	⇒-I, (3.2.6)
3	$(p \wedge q) = (q \wedge p)$	=-I, 1, 2

Rule ⇒-I allows us to conclude $p \Rightarrow q$ if we have a proof of q given premise p. On the other hand, if we take $p \Rightarrow q$ as a premise, then rule ⇒-E allows us to conclude that q holds when p is given. We see that the following relationship holds:

> **Deduction Theorem.** "**Infer** $p \Rightarrow q$" is a theorem of the natural deduction system, which can be interpreted to mean that $p \Rightarrow q$ is a tautology, *iff* "**From** p **infer** q" is a theorem. □

Another example of the use of ⇒-I shows that p implies itself:

(3.3.3) **Infer** $p \Rightarrow p$

1	$p \Rightarrow p$	⇒-I, exercise 2 of section 3.2

Subproofs

A proof can be included within a proof, much the way a procedure can be included within a program. This allows the premise of ⇒-I to appear as a line of a proof. To illustrate this, (3.3.2) is rewritten in (3.3.4) to include proof (3.2.5) as a subproof. The subproof happens to be on line 1 here, but it could be on any line. If the subtheorem appears on line j

(say) of the main proof, then its proof appears indented underneath, with its lines numbered *j*.1, *j*.2, etc. We could have replaced the reference to (3.2.6) by a subproof in a similar manner.

(3.3.4)

Infer $(p \wedge q) = (q \wedge p)$			
1	**From** $p \wedge q$ **infer** $q \wedge p$		
	1.1	p	$\wedge$-E, pr 1
	1.2	q	$\wedge$-E, pr 1
	1.3	$q \wedge p$	$\wedge$-I, 1.2, 1.1
2	$(p \wedge q) \Rightarrow (q \wedge p)$		$\Rightarrow$-I, 1
3	$(q \wedge p) \Rightarrow (p \wedge q)$		$\Rightarrow$-I, (3.2.6)
4	$(p \wedge q) = (q \wedge p)$		$=$-I, 2, 3

Another example of a proof with a subproof is given in (3.3.5). Again, it may be instructive to compare the proof to an English version:

> Suppose $(q \vee s) \Rightarrow (p \wedge q)$. To prove equivalence, we must show also that $(p \wedge q) \Rightarrow (q \vee s)$. [Note how this uses rule $=$-I, that $a \Rightarrow b$ and $b \Rightarrow a$ means $a = b$. These sentences correspond to lines 1, 3 and 4 of the formal proof.] To prove $(p \wedge q) \Rightarrow (q \vee s)$, argue as follows. Assume $p \wedge q$ is true. Then so is q. By the definition of **or**, so is $q \vee s$. [Note the correspondence to lines 2.1-2.2.] □

(3.3.5)

From $(q \vee s) \Rightarrow (p \wedge q)$ **infer** $(q \vee s) = (p \wedge q)$			
1	$(q \vee s) \Rightarrow (p \wedge q)$		pr 1
2	**From** $p \wedge q$ **infer** $q \vee s$		
	2.1	q	$\wedge$-E, pr 1
	2.2	$q \vee s$	$\vee$-I, 2.1
3	$(p \wedge q) \Rightarrow (q \vee s)$		$\Rightarrow$-I, 2
4	$(q \vee s) = (p \wedge q)$		$=$-I, 1, 3

As mentioned earlier, the relationship between proofs and sub-proofs in logic is similar to the relationship between procedures and sub-procedures (modules and sub-modules) in programs. A theorem and its proof can be used in two ways: first, use the theorem to prove something else; secondly, study the proof of the theorem. A procedure and its description can be used in two ways: first, understand the description so that calls of the procedure can be written; secondly, study the procedure body to understand how the procedure works. This similarity should make the idea of subproofs easy to understand.

Scope rules

A subproof can contain references not only to previous lines in its proof, but also to previous lines that occur in surrounding proofs. We call these *global* line references. However, "recursion" is not allowed; a line j (say) may not contain a reference to a theorem whose proof is not finished by line j.

The reader skilled in the use of block structure in languages like PL/I, ALGOL 60 and Pascal will have no difficulty in understanding this scope rule, for essentially the same scope mechanism is employed here (except for the restriction against recursion). Let us state the rule more precisely.

(3.3.6) **Scope rule**. Line i of a proof, where i is an integer, may contain references to lines 1, ..., $i-1$. Line $j.i$, where i is an integer, may contain references to lines $j.1$, ..., $j.(i-1)$ and to any lines referenceable from line j (this excludes references to line j itself). □

Example (3.3.7) illustrates the use of this scope rule; line 2.2 refers to line 1, which is outside the proof of line 2.

(3.3.7)

	From $p \Rightarrow (q \Rightarrow r)$ **infer** $(p \wedge q) \Rightarrow r$		
1	$p \Rightarrow (q \Rightarrow r)$		pr 1
2	**From** $p \wedge q$ **infer** r		
	2.1	p	$\wedge$-E, pr 1
	2.2	$q \Rightarrow r$	$\Rightarrow$-E, 1, 2.1
	2.3	q	$\wedge$-E, pr 1
	2.4	r	$\Rightarrow$-E, 2.2, 2.3
3	$(p \wedge q) \Rightarrow r$		$\Rightarrow$-I, 2

Below we illustrate an invalid use of the scope rule.

	From p **infer** $p \Rightarrow \neg p$ (Proof INVALID)		
1	p		pr 1
2	**From** p **infer** $\neg p$		
	2.1	p	pr 1
	2.2	$p \Rightarrow \neg p$	$\Rightarrow$-I, 2 (invalid reference to line 2)
2	$p \Rightarrow \neg p$		$\Rightarrow$-I, 2 (valid reference to line 2)

We illustrate another common mistake below; the use of a line that is not in a surrounding proof. Below, on line 6.1 an attempt is made to reference s on line 4.1. Since line 4.1 is not in a *surrounding* proof, this is not allowed.

A subproof using global references is being proved *in a particular context*. Taken out of context, the subproof may not be true because it relies

From $p \vee q$, $p \Rightarrow s$, $s \Rightarrow r$ **infer** r (proof INVALID)

1	$p \vee q$		pr 1
2	$p \Rightarrow s$		pr 2
3	$s \Rightarrow r$		pr 3
4	**From** p **infer** r		
	4.1	s	$\Rightarrow$E, 2, pr 1 (valid reference to 2)
	4.2	r	$\Rightarrow$E, 3, 4.1 (valid reference to 3)
5	$p \Rightarrow r$		$\Rightarrow$I, 4
6	**From** q **infer** r		
	6.1	r	$\Rightarrow$E, 3, 4.1 (invalid reference to 4.1)
7	$q \Rightarrow r$		$\Rightarrow$I, 6
8	r		$\vee$-E, 1, 5, 7

on assumptions about the context. This again points up the similarity between ALGOL-like procedures and subproofs. Facts assumed outside a subproof can be used within the proof, just as variables declared outside a procedure can be used within a procedure, using the same scope mechanism.

To end this discussion of scope, we give a proof with two levels of subproof. It can be understood most easily as follows. First read lines 1, 2 and 3 (don't read the the proof of the lemma on line 2) and satisfy yourself that *if* the proof of the lemma on line 2 is correct, then the whole proof is correct. Next, study the proof of the lemma on line 2 (only lines 2.1, 2.2 and 2.3). Finally, study the proof of the lemma on line 2.2, which refers to a line two levels out in the proof.

(3.3.8)

From $(p \wedge q) \Rightarrow r$ **infer** $p \Rightarrow (q \Rightarrow r)$

1	$(p \wedge q) \Rightarrow r$			pr 1
2	**From** p **infer** $q \Rightarrow r$			
	2.1	p		pr 1
	2.2	**From** q **infer** r		
		2.2.1	$p \wedge q$	$\wedge$-I, 2.1, pr 1
		2.2.2	r	$\Rightarrow$E, 1, 2.2.1
	2.3	$q \Rightarrow r$		$\Rightarrow$I, 2.2
3	$p \Rightarrow (q \Rightarrow r)$			$\Rightarrow$I, 2

Proof by contradiction

A proof by contradiction typically proceeds as follows. One makes an assumption. From this assumption one proceeds to prove a contradiction, say, by showing that something is both true and false. Since such a

contradiction cannot possibly happen, and since the proof from assumption to contradiction is valid, the assumption must be false.

Proof by contradiction is embodied in the proof rules $\neg$-I and $\neg$-E:

$$\text{(3.3.9)}\quad \neg\text{-I:}\ \frac{\textbf{From } E \textbf{ infer } El \wedge \neg El}{\neg E}$$

$$\text{(3.3.10)}\quad \neg\text{-E:}\ \frac{\textbf{From } \neg E \textbf{ infer } El \wedge \neg El}{E}$$

Rule $\neg$-I indicates that if "**From** E **infer** $El \wedge \neg El$" has been proved for some proposition El, then one can write $\neg E$ on a line of the proof.

Rule $\neg$-I similarly allows us to conclude that E holds if a proof of "**From** $\neg E$ **infer** $El \wedge \neg El$" exists, for some proposition El.

We show in (3.3.11) an example of the use of rule $\neg$-I, that from p we can conclude $\neg\neg p$.

(3.3.11)

From p **infer** $\neg\neg p$		
1	p	pr 1
2	**From** $\neg p$ **infer** $p \wedge \neg p$	
	2.1 $p \wedge \neg p$	$\wedge$-I, 1, pr 1
3	$\neg\neg p$	$\neg$-I, 2

Rule $\neg$-I is used to prove that $\neg\neg p$ follows from p; similarly, rule $\neg$-E is used in (3.3.12) to prove that p follows from $\neg\neg p$.

(3.3.12)

From $\neg\neg p$ **infer** p		
1	$\neg\neg p$	pr 1
2	**From** $\neg p$ **infer** $\neg p \wedge \neg\neg p$	
	2.1 $\neg p \wedge \neg\neg p$	$\wedge$-I, pr 1, 1
3	p	$\neg$-E, 2

Theorems (3.3.11) and (3.3.12) look quite similar, and yet both proofs are needed; one cannot simply get one from the other more easily than they are proven here. More importantly, both of the *rules* $\neg$-I and $\neg$-E are needed; if one is omitted from the proof system, we will be unable to deduce some propositions that are tautologies in the sense described in section 1.5. This may seem strange, since the rules look so similar.

Let us give two more proofs. The first one indicates that from p and $\neg p$ one can prove *any* proposition q, even one that is equivalent to false. This is because both p and $\neg p$ cannot both be true at the same time, and hence the premises form an absurdity.

(3.3.13)

From $p, \neg p$ **infer** q			
1	p		pr 1
2	$\neg p$		pr 2
3	**From** $\neg q$ **infer** $p \wedge \neg p$		
	3.1	$p \wedge \neg p$	$\wedge$-I, 1, 2
4	q		$\neg$-E, 3

(3.3.14)

From $p \wedge q$ **infer** $\neg(p \Rightarrow \neg q)$			
1	$p \wedge q$		pr 1
2	**From** $p \Rightarrow \neg q$ **infer** $q \wedge \neg q$		
	2.1	p	$\wedge$-E, 1
	2.2	q	$\wedge$-E, 1
	2.3	$\neg q$	$\Rightarrow$-E, pr 1, 2.1
	2.4	$q \wedge \neg q$	$\wedge$-I, 2.2, 2.3
3	$\neg(p \Rightarrow \neg q)$		$\neg$-I, 2

For comparison, we give an English version of proof (3.3.14). Let $p \wedge q$ be true. Then both p and q are true. Assume that $p \Rightarrow \neg q$ is true. Because p is true this implication allows us to conclude that $\neg q$ is true, but this is absurd because q is true. Hence the assumption that $p \Rightarrow \neg q$ is true is wrong, and $\neg(p \Rightarrow \neg q)$ holds.

Summary

The reader may have noticed a difference between the natural deduction system and the previous systems of evaluation and equivalence transformation: the natural deduction system does not allow the use of constants T and F! The connection between the systems can be stated as follows. If "**Infer** e" is a theorem of the natural deduction system, then e is a tautology and $e = T$ is an equivalence. On the other hand, if $e = T$ is a tautology and e does not contain T and F, then "**Infer** e" is a theorem of the natural deduction system. The omission of T and F is no problem because, by the rule of Substitution, in any proposition T can be replaced by a tautology (e.g. $b \vee \neg b$) and F by the complement of a tautology (e.g. $b \wedge \neg b$) to yield an equivalent proposition.

We summarize what a proof is as follows. A proof of a theorem "**From** $e_1, \cdots, e_n$ **infer** e" or of a theorem "**Infer** e" consists of a sequence of lines. The first line contains the theorem. If the first line is unnumbered, the rest are indented and numbered 1, 2, etc. If the first line has the number i, the rest are indented and numbered $i.1$, $i.2$, etc. The last line must contain proposition e. Each line i must have one of the following four forms:

Form 1: (*i*) e_j pr j

where $1 \leq j \leq n$. The line contains premise j.

Form 2: (*i*) p *Name*, $ref_1, \ldots, ref_q$

Each ref_k either (1) is a line number (which is valid according to scope rule (3.3.6)), or (2) has the form "pr j", in which case it refers to premise e_j of the theorem, or (3) is the name of a previously proven theorem. Let r_k denote the proposition or theorem referred to by ref_k. Then the following must be an instance of inference rule *Name*:

$$\frac{r_1, \cdots, r_q}{p}$$

Form 3: (*i*) p *Theorem name*, $ref_1, \ldots, ref_q$

Theorem name is the name of a previously proved theorem; ref_k is as in Form 2. Let r_k denote the proposition referred to be ref_k. Then "**From** $r_1, \cdots, r_q$ **infer** p" must be the named theorem.

Form 4: (*i*) [Proof of another theorem]

That is, the line contains a complete subproof, whose format follows these rules.

Figure 3.3.1 contains a list of the inference rules.

Historical Notes

The style of the logical system defined in this chapter was conceived principally to capture our "natural" patterns of reasoning. Gerhard Gentzen, a German mathematician who died in an Allied prisoner of war camp just after World War II, developed such a system for mathematical arguments in his 1935 paper *Untersuchungen ueber das logische Schliessen* [20], which is included in [43].

Several textbooks on logic are based on natural deduction, for example W.V.O. Quine's book *Methods of Logic* [41].

The particular block-structured system given here was developed using two sources: *WFF'N PROOF: The Game of Modern Logic*, by Layman E. Allen [1] and the monograph *A Programming Logic*, by Robert Constable and Michael O'Donnell [7]. The former introduces the deduction system through a series of games; it uses prefix notation, partly to avoid problems with parentheses, which we have sidestepped through informality. *A Programming Logic* describes a mechanical program verifier for

PL/CS (a subset of PL/C, which is a subset of PL/1), developed at Cornell University. Its inference rules were developed with ease of presentation and mechanical verification in mind. Actually, the verifier can be used to verify proofs of programs, and includes not only the propositional calculus but also a predicate calculus, including a theory of integers and a theory of strings.

$\wedge$-I: $\dfrac{E_1, \ldots, E_n}{E_1 \wedge \ldots \wedge E_n}$ $\quad$ $\wedge$-E: $\dfrac{E_1 \wedge \ldots \wedge E_n}{E_i}$

$\vee$-I: $\dfrac{E_i}{E_1 \vee \ldots \vee E_n}$ $\quad$ $\vee$-E: $\dfrac{E_1 \vee \ldots \vee E_n, E_1 \Rightarrow E, \ldots, E_n \Rightarrow E}{E}$

$\neg$-I: $\dfrac{\textbf{From } E \textbf{ infer } E1 \wedge \neg E1}{\neg E}$ $\quad$ $\neg$-E: $\dfrac{\textbf{From } \neg E \textbf{ infer } E1 \wedge \neg E1}{E}$

$=$-I: $\dfrac{E1 \Rightarrow E2,\ E2 \Rightarrow E1}{E1 = E2}$ $\quad$ $=$-E: $\dfrac{E1 = E2}{E1 \Rightarrow E2,\ E2 \Rightarrow E1}$

$\Rightarrow$-I: $\dfrac{\textbf{From } E_1, \ldots, E_n \textbf{ infer } E}{(E_1 \wedge \ldots \wedge E_n) \Rightarrow E}$ $\quad$ $\Rightarrow$-E: $\dfrac{E1 \Rightarrow E2,\ E1}{E2}$

Figure 3.3.1 The Set of Basic Inference Rules

Exercises for Section 3.3

1. Use lemma (3.2.11) and inference rule $\Rightarrow$-I to give a 1-line proof that $(p \wedge q \wedge (p \Rightarrow r)) \Rightarrow (r \vee (q \Rightarrow r))$.

2. Prove that $(p \wedge q) \Rightarrow (p \vee q)$, using rule $\Rightarrow$-I.

3. Prove that $q \Rightarrow (q \wedge q)$. Prove that $(q \wedge q) \Rightarrow q$. Use the first two results to prove that $q = (q \wedge q)$. Then rewrite the last proof so that it does not refer to outside proofs.

4. Prove that $p = (p \vee p)$.

5. Prove that $p \Rightarrow ((r \vee s) \Rightarrow p)$.

6. Prove that $q \Rightarrow (r \Rightarrow (q \wedge r))$.

7. Prove that from $p \Rightarrow (r \Rightarrow s)$ follows $r \Rightarrow (p \Rightarrow s)$.

8. What is wrong with the following proof?

Infer $a \Rightarrow b$ (Proof INVALID)

1	a		pr 1
2	**From** $\neg b$ **infer** $b \wedge \neg b$		
	2.1	$\neg b$	pr 1
	2.2	$\neg b \Rightarrow b \wedge \neg b$	$\Rightarrow$-I, 2
	2.3	$b \wedge \neg b$	$\Rightarrow$-E, 2.2, 2.1
2	b		$\neg$-E, 2

9. Prove that from $\neg p$ and $(\neg p \Rightarrow q) \vee (p \wedge (r \Rightarrow q))$ follows $r \Rightarrow q$.

10. Prove that $q \Rightarrow (p \wedge r)$ follows from $q \Rightarrow p$ and $q \Rightarrow r$.

11. Prove that from $\neg q$ follows $q \Rightarrow p$.

12. Prove that from $\neg q$ follows $q \Rightarrow \neg p$.

13. Prove that from $\neg q$ follows $q \Rightarrow (p \wedge \neg p)$.

14. Prove that from $p \vee q$, $\neg q$ follows p.

15. Prove $p \wedge (p \Rightarrow q) \Rightarrow q$.

16. Prove $((p \Rightarrow q) \wedge (q \Rightarrow r)) \Rightarrow (p \Rightarrow r)$.

17. Prove $(p \Rightarrow q) \Rightarrow ((p \wedge \neg q) \Rightarrow q)$.

18. Prove $((p \wedge \neg q) \Rightarrow q) \Rightarrow (p \Rightarrow q)$. [This, together with exercise 17, allows us to prove $(p \Rightarrow q) = ((p \wedge \neg q) \Rightarrow q)$.]

19. Prove $(p \Rightarrow q) \Rightarrow ((p \wedge \neg q) \Rightarrow \neg p)$.

20. Prove $((p \wedge \neg q) \Rightarrow \neg p) \Rightarrow (p \Rightarrow q)$. [This, together with exercise 19, allows us to prove $(p \Rightarrow q) = ((p \wedge \neg q) \Rightarrow \neg p)$.]

21. Prove that $(p = q) \Rightarrow (\neg p = \neg q)$.

22. Prove that $(\neg p = \neg q) \Rightarrow (p = q)$. [This, together with exercise 21, allows us to prove $(p = q) = (\neg p = \neg q)$.]

23. Prove $\neg(p = q) \Rightarrow (\neg p = q)$

24. Prove $(\neg p = q) \Rightarrow \neg(p = q)$. [This, together with exercise 21, allows us to prove the law of Inequality, $\neg(p = q) = (\neg p = q)$.]

25. Prove $(p = q) \Rightarrow (q = p)$.

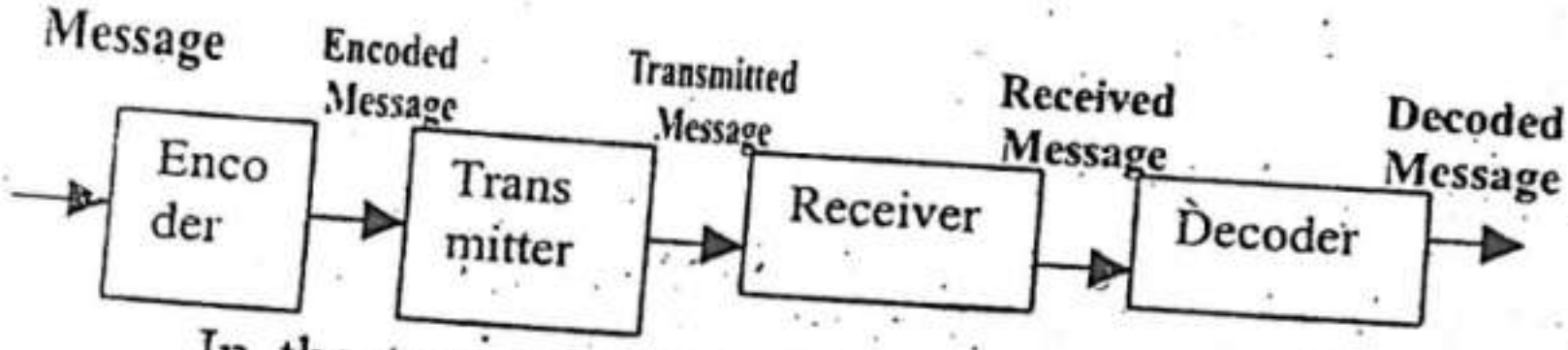
26. Use a rule of Contradiction to prove **From** p **infer** p.

27. For each of the proofs of exercise 1-7, 9-25, give a version in English. (It need not follow the formal proof exactly.)

LESSON – 2

CODING THEORY

INTRODUCTION



In the transmission of information we assume that we are concerned with the transmission of some unit called a **Message.** This message is first encoded in a form suitable for transmission, and then processed by a transmitter and sent across a channel. A receiver then picks up the information coming across the channel. This information is then decoded by a decoder which transforms the information into a received message.

Example:-

A Radio broadcast takes a message in English language and transmits it across space to a receiver which then converts the radiowaves into sound waves.

Example:-

Visual input into the Retina, which consists of patterns of photons hitting the retina. These patterns are encoded into electric impulses in some of the cells affected by the incoming photons. These impulses are transmitted across a network of neurons and received in a visual area of the brain, where the visual pattern is recognized.

Two aspects of transmission of information

Ensuring the secrecy of a Transmitted message

1) During the transmission of a message across a channel there may be numerous opportunities for

interception.(ex. , the interception of a radio transmission by a radio receiver). If the message can be comprehended by the interceptors, then the secrecy is lost. Thus an encoding process is often used which makes the message unintelligible to an unintended receiver. After this encoded message is received it must be decoded in order for the receipient to understand the transmitted information.

The study of encoding and decoding to ensure secrecy is called CRYPTOGRAPHY.

2) Another important aspect of information transmission is concerned with reception and decoding in the presence of unreliable transmission. This unreliability of transmission may be due to noise in the channel or to very weak signals.

CRYPTOGRAPHY

Deciphering:- It is the process of discovering the decoding procedure when only the transmitted message is available. An original message is known as the **plaintext** while the coded message is called the **ciphertext**. The process of converting from plaintext to ciphertext is known as **enciphering** or **encryption**.

Definition:- An **Encoding** Φ of a set A into the set X is a one-to-one function φ from A into X.

$$\Phi : A \rightarrow X$$

If $a_1, a_2 \varepsilon A$ then $\Phi(a_1) = \Phi(a_2)$

$$a1 \rightarrow \Phi(a_1)$$

$$a2 \rightarrow \Phi(a_2)$$

Therefore encoding Φ is extended to the set of all sequences $a_1a_2 \ldots a_n$ where a_i is contained in A and is given by $\mathbf{a_1a_2a_3 \ldots . a_n \rightarrow \Phi(a_1)\ \Phi(a_2) \ldots\ \Phi(a_n)}$

Definition:- The **Decoding** of Φ is defined to be the function Φ^{-1} from $\Phi(A)$ to A.

Therefore decoding Φ^{-1} is extended to the set of all sequences $x_1x_2...x_n$ where x_i is in $\Phi(A)$ which is a subset of X and is given by $x_1x_2...x_n \rightarrow \Phi^{-1}(x_1)\ \Phi^{-1}(x_2)...\ \Phi^{-1}(x_n)$

Monoalphabetic:-

We generally take A to be one of the sets

- { the set of the English alphabet }
- { n-tuples of letters of English alphabet with 'n' a small positive integer }
- Z / (k) with k=26, 29 or 2

If A is a set of letters of a standard alphabet or Z / (k) then the encoding is called Monoalphabetic.

Caesar Cypher Method:-

Approximately 2000 years ago, Julius Caesar used monoalphabetic encoding to conceal the information contained in some of his written communications. This encoding is called a Caesar Cypher and Φ(letter) is the letter three letters after a given letter.

The correspondence between the alphabet and Z/(26)

A	B	C	D	E	F	G	H
	I	J	K	L			
1	2	3	4	5	6	7	8
	9	10	11	12			

M	N	O	P	Q	R	S	T
	U	V	W	X			
13	14	15	16	17	18	19	20
	21	22	23	24			

Y	Z
25	26

Example:-

Plaintext: Is is not our level of prosperity that makes for happiness.

Encoding:

9	20	9	19	14	15	20
15	21	18	12	5	22	5
12	15	6	16	18	15	19
16	5	18	9	20	25	20
8	1	20	13	1	11	5
19	6	15	18	8	1	16
16	9	14	5	19	19.	

Then

12	23	12	22	17	18	23
18	24	21	15	8	25	8
15	18	9	19	21	18	22
19	8	21	12	23	2	23
11	4	23	16	4	14	8
22	9	18	21	11	4	19
19	12	17	8	22	22	

Therefore

LWLYQRW RXUOHYHO RISURVSH ULWBWKDW PDNHV IRUKDSS LQHVV

Which is the encoded message. This Caesar encoding can be given by

$\Phi([x]) = [x] + [3]$ where addition takes place in $Z/(26)$

Mathematical formulation:-

If a and b are integers with gcd{a, 26} = 1, then define a Modular Encoding

$\Phi([x]) = [a][x]+[b]$

The Caesar Cypher is defined to be a modular encoding with a=1. The gcd{a, 26} must be equal to 1, since otherwise the modular encoding would not be one-to-one. $\Phi(0)=b$ and $\Phi(26 / \gcd\{a,26\}) = [b]$

If gcd{a, 26}=1, then a^{-1} exists and there is a decoding $\Phi^{-1}(y)=[a]^{-1}y-[a]^{-1}[b]$

The element $[a]^{-1}$ can be determined by the Euclidean Algorithm. The encoding Φ and the decoding Φ^{-1} give all the information needed to encode a message and decode an encoded message.

Procedure for decoding:-

If it is suspected that the code is a Caesar Cypher, then simply take the intercepted encoded word and construct a table as follows:

Example:-

X	T	W	T	E	L	C	J
	encoded message						
Y	U	X	U	F	M	D	K
Z	V	Y	V	G	N	E	L
A	W	Z	W	H	O	F	M
B	X	A	X	I	P	G	N
C	Y	B	Y	J	Q	H	O
D	Z	C	Z	K	R	I	P
E	A	D	A	L	S	J	Q
F	B	E	B	M	T	K	R
G	C	F	C	N	U	L	S
H	D	G	D	O	V	M	T
I	E	H	E	P	W	N	O
J	F	I	F	Q	X	O	V
K	G	J	G	R	Y	P	W
L	H	K	H	S	Z	Q	X
M	I	L	I	T	A	R	Y
	ecoded word						
N	J	M	J	U	B	S	Z
O	K	N	K	V	C	T	A
P	L	O	L	W	D	U	B
Q	M	P	M	X	E	V	C
R	N	Q	N	Y	F	W	D
S	O	R	O	Z	G	X	E

T	P	S	P	A	H	Y	F
U	Q	T	Q	B	I	Z	G
V	R	U	R	C	J	A	H
W	S	V	S	D	K	B	I

The first row of this table consists of the intercepted encoded word and underneath each letter of the encoded word is a column which continues the alphabet from that letter on. The rows of the table are scanned to see if any word is recognized.

In the example, we suppose that XTWTELCJ is decoded as "military" and that this encoding is an example of a Caesar Cypher in which the displacement is 11 letters.

Hence the decoding is $\Phi^{-1}([y]) = [y] + 15$

However there are some means to determine $\Phi^{-1}([y])$ for several different values of y, then the system of equations

$$y_1z_1 + z_2 = \Phi^{-1}(y_1)$$
$$y_2z_1 + z_2 = \Phi^{-1}(y_2)$$

can possibly be solved for z_1 and z_2 in $Z/(2b)$

where $z_1 = [a]^{-1}$ and $z_2 = -[a]^{-1}b$

$(y_1 - y_2)\, z = \Phi^{-1}(y_1) - \Phi^{-1}(y_2)$

This has a solution if and only if gid $\{(y_1 - y_2), 2b)\}$ divides $\Phi^{-1}(y_1) - \Phi^{-1}(y_2)$

Therefore in this case there are gid $\{(y_1 - y_2), 2b)\}$ possibilities for z_1 one of which will yield the correct decoding equation.

Determining $\Phi^{-1}[y]$ for some values of [y]

An extremely valuable tool in this search is a standard frequency count of the occurrence of the letters of the English alphabet. This count is made visually striking

by putting the letters of the alphabet in a row and beneatl each letter; a horizontal stroke for each occurrence of tha letter in the message.

*The most frequently occurring letter of th encoding text stands for the letter E.

* The next most frequently occurring letter ma stand for the letter T.

This gives 2 equations in 2 unknowns which can be solved and a portion of the text can be decoded to see if the calculated values for a and b give a sensible decoding.

The standard frequency count of letters occurring in newspaper text is given below:

A	7.3	H	3.5	O	7.4	U	2.7
B	0.9	I	7.4	P	2.7	V	1.3
C	3.0	J	0.2	Q	0.3	W	1.6
D	4.4	K	0.3	R	7.7	X	0.5
E	13.0	L	3.5	S	6.3	Y	1.9
F	2.8	M	2.5	T	9.3	Z	0.1
G	1.6	N	7.8				

Example(*):- An intercepted coded Message

KMIUT	RKJKM	MWMMU
WXPQK	XNWUK	
TWYIP	FJWMD	WQPPA
CWXWJ	KPIXC	
WRWQP	JIQDA	YWJZJ
AUKRD	WJXKP	
WWXWJ	CEMAS	JQWM

The message is written in blocks of 5 letters.

The frequency count for the code

Let us assume that the most frequently occurring letter is W. Therefore $\Phi^{-1}(W)=E$

One possible way is by trial and error method, assuming that $\Phi^{-1}(J)=T$ and solving the set of equations and then attempting to decode the message, to see if we actually have the proper solution. If not, we then try $\Phi^{-1}(K)=T$ and repeat the procedure.

Digraphs:-

In English language, there are certain pairs of letters that occur more frequently than other pairs of letters. Adjacent pairs of letters in the English language are called digraphs.

Most frequent digraphs (out of 80,000 characters of text)

TH	2161	AN	1216	TI	865
HE	2053	EN	1029	OR	861
IN	1550	AT	1019	ST	823
ER	1436	ES	917	AR	764
RE	1280	ED	890	ND	761
ON	1232	TE	872	TO	756

A count of the digraphs in the code in Example(*) that begin with the letter W gives the following frequency distribution.

Digraph frequency for W- in Example(*)

	J	M	Q	R	U	W	X	Y
W	||||	|||	||	|	|	|	|||	|

Modular Decoding:-

A Modular code has the advantage that it is relatively simple to encode and decode. It has the advantage that knowledge of how two letters are decoded is sufficient to determine how all letters are decoded. Thus to decode a message requires only that the modular integers z_1 and z_2 be known. It is much more difficult to devise a decoding method for arbitrary encoding functions Φ. In this case the receiver of the encoded message must have the decoding function Φ^{-1} available in tabular form in a paper, in the form of a specially constructed decoding machine or in memorized form. Since written form or a specially constructed machine constitutes physical evidence, often it is desirable to have decoding information in as concise a form as possible. This is one advantage of modular decoding.

Key-word encoding:-

Another type of monoalphabetic encoding requiring minimal information on the part of the decoder is key-word encoding. List the alphabets on a line. Write the key word below the alphabet with the first letter of the key word corresponding to A, the second to B and so on, with repeated letters omitted. After the keyword, continue writing the alphabet but omit the letters occurring in the key word. This gives the encoding correspondence, hence the decoding correspondence.

Example:-

Keyword is "Quebec liberation"

Keyword encoding is

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Q	U	E	B	C	L	I	R	A	T	O	N	D	F	G	H	J	K	M	P	S	V	W	X	Y	Z

Matrix Encoding (Block encoding)

The alphabet is represented by the integers modulo 26. The Base set of this encoding is the vector space of m-tuples with elements in the integers modulo 26.Thus A can be considered as block of 'm' letters.

Let M be an invertible m x m Matrix with entries in Z/ (26). The encoding thus takes a block of M letters , realizes this as a vector in $[Z/26]^m$ multiplies this vector by M and then produces the corresponding block of letters.

Example:- Let $M = \begin{bmatrix} 1 & 1 \\ 13 & 12 \end{bmatrix}$ then the word "to" becomes

$$[20,15] \begin{bmatrix} 1 & 1 \\ 13 & 12 \end{bmatrix} = [7, 18] = GR$$

Example:- Let $M = \begin{bmatrix} 1 & 1 \\ 13 & 12 \end{bmatrix}$ then the word "do" becomes

$$[4,15] \begin{bmatrix} 1 & 1 \\ 13 & 12 \end{bmatrix} = [17, 2] = QB$$

If the matrix M^{-1} is calculated, then the encoding is (v M) $M^{-1} = v$ for each vector v.

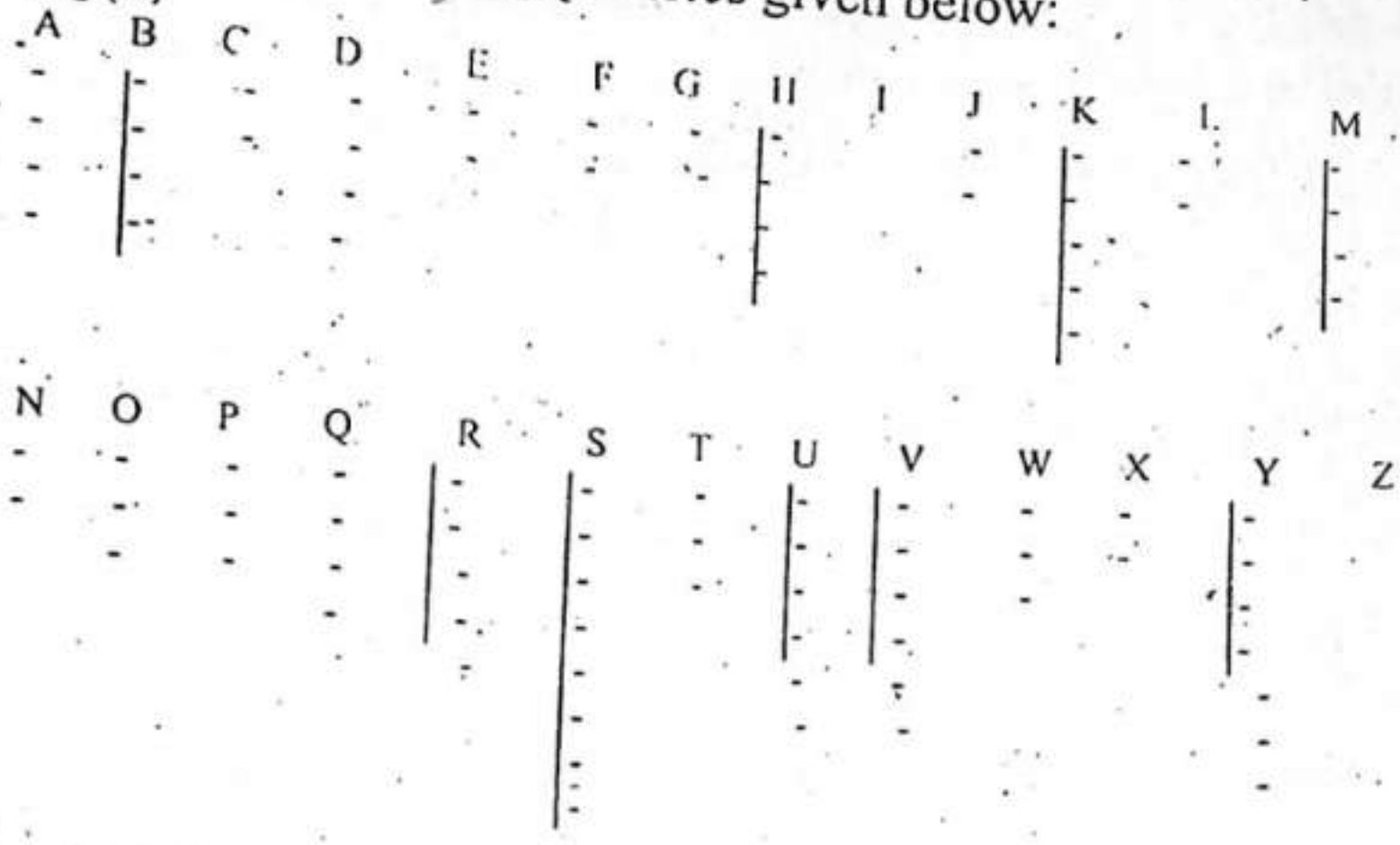
Example:- $M^{-1} = \begin{bmatrix} 14 & 1 \\ 13 & 25 \end{bmatrix}$

$$[QB]\ [17\ 2] \begin{bmatrix} 14 & 1 \\ 13 & 25 \end{bmatrix} = [4\ 15] = [D\ O]$$

'f the encoding and decoding matrices are unknown, but ;amples of encoded messages are available, then the 'ollowing procedure can be used:

1) Guess whether the code is monoalphabetic or not This can be done by considering the frequency distributior of the code letters. A monoalphabetic code has grea variation in the frequencies of individual letters whicl should match the frequencies given below:

Fig (A)



A matrix encoding has a more even frequency distribution of the letters.

2) Once it has been decided that the code is not monoalphabetic, and that it is possibly a matrix encoding, the next step is to determine the BLOCK SIZE(that is, the dimension of the vector space). If the complete message has been intercepted, then the total number of characters can be counted. This is an integer value(multiple) of a certain number of blocks. Therefore the block size is a divisor of the total number of characters.

Example:-

Message 1:-

LMRSX	CCHDV	SHBUS	NBTGU	RDKDV
WLUQE	MQKYK	YHWSA	KYURG	SPRYV
WQPTT	UYM			

Message 2:-

OXJRM	FURYK	AGDVS	HBUPH	OBKYH
WSAKY	SBUVS	FKVQT	MEOTV	M

The frequency count is given in Fig (A)

The frequency count is much more evenly distributed than the standard frequency count. Hence we assume that the code is not monoalphabetic and that it is possibly a matrix encoding.

The I message contains 58 characters. Hence the block length is 2, 29 or 58.

The II message contains 46 characters. Hence the block length is 2, 23, 46.

Hence, if the code is matrix encoding then the block size is 2.

Then the frequencies of the diagraphs corresponding to vectors can be counted and a correspondence between frequent diagraphs and frequent digraphs in the English language can be attempted.

If $\Phi^{-1}(V)$ and $\Phi^{-1}(W)$ are known for vectors V and W in $[Z/26]^2$ then the system of 4 equations and 4 unknowns given by

$VM^{-1} = \Phi^{-1}(V)$

$WM^{-1} = \Phi^{-1}(W)$ can be solved. However, the code breaker may have other information available which may be helpful in cracking the code.

For example, he may know the general nature of the message hence he may except the occurrence of certain phrases or certain words. Thus if strings of characters are repeated, this may indicate a possible decoding.

In this above example, the code breaker may suspect the word "Indochina" occurs in the text. This words begins with IN and 6 characters later repeats the same word

IN. Also, in the I message the sequence "KYHWSAKY" occurs and a similar sequence occurs in the II message. Hence assume

$$\Phi^{-1}(KY) = IN \text{ and}$$
$$\Phi^{-1}(HW) = DO.$$

Thus $[[11], [25]] \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix} = [[9], [14]]$

$[[8], [23]] \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix} = [[4], [15]]$

where $\begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix}$ is the inverse of the encoding matrix.

$$11x_1 + 25x_3 = 9 \quad \text{----(1)}$$
$$8x_1 + 23x_3 = 4 \quad \text{----(2)}$$
$$11x_2 + 25x_4 = 9 \quad \text{----(3)}$$
$$8x_2 + 23x_4 = 4 \quad \text{----(4)}$$

On solving the above equations, we get $x_1 = 3$, $x_2 = 25$, $x_3 = 24$, $x_4 = 1$.

(i.e) $\begin{bmatrix} 3 & 25 \\ 24 & 1 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix}$

which is the inverse of the encoding matrix. Consider the first 8 letters of the I message.

LMRSXCCH

$\Phi^{-1}(LM) = \Phi^{-1}[12,13] = [12\ 13] \begin{bmatrix} 3 & 25 \\ 24 & 1 \end{bmatrix} = [10\ 1] = JA$

$\Phi^{-1}(RS) = \Phi^{-1}[18\ 19] = [18\ 19] \begin{bmatrix} 3 & 25 \\ 24 & 1 \end{bmatrix} = [16\ 1] = PA$

$\Phi^{-1}(XC) = \Phi^{-1}[24\ 3] = [24\ 3] \begin{bmatrix} 3 & 25 \\ 24 & 1 \end{bmatrix} = [14\ 5] = NE$

$\Phi^{-1}(CH) = \Phi^{-1}[3\ 8] = [3\ 8] \begin{bmatrix} 3 & 25 \\ 24 & 1 \end{bmatrix} = [19\ 5] = SE$

Therefore Φ^{-1} (LMRSXCCH) = JAPANESE

Similarly, consider the remaining words.

DV SH BU SN BT GU RD KD VW LU QE
MQ KY KY HW SA KY UR GS PR YV WQPT
TU YM.

$\Phi^{-1}(DV) = \Phi^{-1}[4\ 22] = [4\ 22] \begin{bmatrix} 3 & 25 \\ 24 & 1 \end{bmatrix} = [20\ 18] = TR$

$\Phi^{-1}(SH) = \Phi^{-1}[19\ 8] = [15\ 15] = OO$.....

Decoded the remainder of the encoded text gives the following messages.

"Japanese troops currently stationed in Indochina will be withdrawn"

"Withdrawal of troops from Indochina is a smokescreen".

Deciphering an encoded message encoded by matrix multiplication of vectors of large block size is extremely difficult. However for block size 2 or 3 the problem is traceable. One drawback is. if a single error

occurs in a block during the handling of an encoded message, then that error will affect the decoding of every character in that block. For block size 2 or 3 , this is not a serious problem, since much of the message can still be constructed from the context. But for large block size a single error results in a decoding that is unreadable in that block.

Scrambled codes

Permutations are the basis for a type of cryptographic code called a scrambled code. If Π is a permutation on {1, 2, 3, ...n }, then we define the encoding function Π to be the map from $(a_1, a_2, \ldots a_n)$ to $(a_{\Pi(1)}, \ldots a_{\Pi(n)})$. This is a very special type of matrix encoding, since the mapping is linear.

Example:- Π = (1 3 2 5 4) and we wish to use the encoding induced by their permutation to encode the message, "THE TRIAL BEGINS TOMORROW".

The message is divided into blocks of five and to encode a block the II (1) letter is placed in the first position, the II(2) letter in the second position, the II(3) letter in the third position, the II (4) letter in the 4th position, and the II (5) letter in the fifth position.

The blocked message is

THETR IALBE GINST OMORR OWZZZ

Which is encoded as

ERHTT LEAIB NTIQS ORMOR ZZWOZ

Decoding a Scrambled code:-

First make sure that the encoding is a permutation encoding. The frequency count of the occurrences of letters

in a permutation encoding yields a frequency distributio very similar to a standard distribution. Thus one ca assume that the letters in the original message ar permuted.

Secondly, determine the block size. If a sequence o letters is repeated, then it can be assumed that the sam word or phrase was repeated in the original message. Henc the block size is a divisor of the number of letter separating the successive occurrences of the repeate sequence.

Third factor is determine the permutation, Once th block size is determined, construct a table whose ith row i the ith block. Permute the columns of the table to obtain a reasonable fit of pairs and continue until the message car be read from the table.

Example:-

SAIAL	GEINR	EELOA	MHDSE
YNAAA	SSYSA	IRLFE	IREOC
WLSIP	LLUBC	LATMK	OTAIL
ASPSZ			

A frequent count performed on this message and it is compared to the standard frequency count. Since the frequency of occurrence of letters in the message so closely matches the standard count, we assume that the code is a permutation code.

Standard frequency count

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

The sequence SAIRL is repeated with an interval of 28 letters from the start of the first occurrence to the start of the second occurrence. Thus the block size of the encoding is a divisor of 28. Hence it must be 2, 4, 7, or 28.

A quick check shows that the block length is not 2. Suppose that the block length is 4. Write the message in a table with 4 columns, where row i is simply block i of the message. Then permute the columns, we will eventually obtain the message.

(1)	(2)	(3)	(4)
S	A	I	R
L	G	E	I
N	R	E	E
L	O	A	M
H	D	S	E
Y	N	A	A
A	S	S	Y
S	A	I	R
L	F	E	I
R	E	O	C
W	L	S	I
P	L	L	U
B	C	L	A
T	M	K	O
A	S	P	S
Z.			

Then permuting the columns

(3)	(1)	(5)	(2)
I	S	R	A
E	L	I	G
E	N	E	R
A	L	M	O
S	H	E	D
A	Y	A	N
S	A	Y	S
I	S	R	A
E	L	I	F
O	R	C	E
S	W	I	L
L	P	U	L
L	B	A	C
K	T	O	M
I	T	L	A
P	A	S	S

which can be read as:

"Israeli general Moshe Dayan says Israeli forces will pull back to Mitla Pass."

This method works only if the block length is relatively short compared to the length of the total message.

II type (completely filled Rectangle Code)

One kind of scrambled code having the block length equal to the message length is a completely filled rectangle code. To encode a message using this method choose a rectangle width k and place the integers from 1 through k in the top row of a table, in arbitrary order. Write the message in blocks of k letters underneath the first row to form a rectangle. The encoded message is now written out by first writing the column labeled 1 horizontally, followed by the column labeled 2, and continuing until the final column k is written horizontally. For example, the message, "The Chilean junta wishes to meet," can be encoded by the following table.

2	1	3
T	H	E
C	H	I
L	E	A
N	J	U
N	T	A
W	I	S
H	E	S
T	O	M
E	E	T

The message then becomes

HHEJTIEOE
TCLNNWHTE
EIAUASSMT,

which is written

HHEJT IEOET CLNNW WHTE AUASS MT.

To decode a message encoded by a completely filled rectangle, the dimension of the rectangle must be determined. If the rectangle is completely filled, then the length multiplied by the width will be the number of letters in the message. Hence the length and width of the rectangle are factors of the number of letters in the message. Once the the dimensions of the rectangle are decided, fill in the

rectangle with the encoded message by writing the encoded message vertically. Once this is done, permute the columns until the message can be read.

As an example, suppose the message

SCOPI	NUDO	TSERR	YAPN	HUVMF
OCZLT	ROT	EREEF	TAEL	PSRIF
ELNUR	EHGE	MATTE	OIAS	—

is intercepted. A frequency count indicates that the message is a scrambled code, and we can see if it is a completely filled rectangle encoding. Since there are 70 letters in the message, the rectangle could be 35 by 2, 14 by 5 or 10 by 7. It is easily checked that the encoding is not a 35 by 2 completely filled rectangle encoding, so the size 14 by 5 is next tried. The decoding rectangle is given in below.

S	R	L	A	E
C	N	T	E	H
O	Y	O	L	G
P	A	A	P	E
I	P	R	S	M
T	N	O	R	A
N	H	T	I	T
U	U	E	F	T
D	V	R	E	E
O	M	E	L	P
T	F	E	N	O
S	O	F	U	I
E	C	N	R	A
R	Z	T	O	S

The columns of the table should now be permutted until a message is formed. If we examine the third row, a permutation gives the suffix OLOGY and, if we try this ordering on the columns we obtain.

S	A	L	E	R
C	E	T	H	N
O	L	O	G	Y
P	P	A	E	A
I	S	R	M	P

or

L	A	S	E	R
T	E	C	H	N
O	L	O	G	Y
A	P	P	E	A
R	S	I	M	P

T	R	O	A	N
N	I	T	T	H
U	F	E	T	U
D	E	R	E	V
O	L	E	P	M
T	N	E	O	F
S	U	F	I	O
E	R	N	A	C
R	O	T	S	Z

O	R	T	A	N
T	I	N	T	H
E	F	U	T	U
R	E	D	E	V
E	L	O	P	M
E	N	T	O	P
F	U	S	I	O
N	R	E	A	O
T	O	R	S	Z

The message now reads:

Laser technology appears important in the future development of fusion reactorsz.

As an example suppose the following message is intercepted and is suspected to contain the phrase "heavy water".

HWALA	AAETY	LEDLT	CVPRE	GNAHT
TCDPD	AARAT	UILEE	SAOOG	YEANO
TSNB.				

Thus the procedure is to block the suspected word into appropriate widths and examine the resulting columns to see if these combinations of letters occur. This yields the width of the rectangle and, in our example located somewhere in the message are letter sequences corresponding to the columns in one of the following,.

HEAVYWATE
R

HEAVYWAT ER	HEAVYWA TER	HEAVYW ATER	HEAVY WATER	HEAV YWAT ER	HEA VYW ATE R	HE AV YW AT R

Corresponding to rectangle widths 9. 8, 7, 6, 5, 4, 3, and 2 respectively.

Thus the final incompletely filled rectangle is

	2	1	7	4	6	5
3	T	H	E	H	E	A
V	Y	W	A	T	E	R
P	L	A	N	T	S	A
R	E	L	O	C	A	T
E	D	A	T	D	O	U
G	L	A	S	P	O	I
N	T	A	N	D	G	L
A	C	E	B	A	Y	

Which yields the message:

The heavy-water plants are located at Douglas Point and Glace Bay.

In general, the deciphering of an incompletely filled rectangular encoding is difficult, but an analysis of the diagraphs that can possibly occur often helps in starting the decipherment. For example, if the letter Q occurs in the encoded message, then in the decoding rectangle it will most likely be followed by U, which may indicate a possible start in the decipherment.

THE HAMMING METRIC

In the last three sections dealing with cryptography, the encoding function Φ mapped the base set A in a one-to-one fashion onto the base set A. However, for the error

detection and error correction aspects of coding theory, the function Φ must map the base set properly into the image set X. If $\Phi(A) = X$, and an error in transmission occurs, then the received word is still an element in $\Phi(A)$, hence will be decoded improperly. However, if $\Phi(A)$ is a proper subset of X, then quite possibly $\Phi(a)$ could be transmitted and received as x, an element not in $\Phi(A)$. Thus someone receiving the transmission would know there was an error in the transmission.

Usually data, whether occurring as deep-space communication or communication between earthbound computers, is coded in binary. Thus the messages are sequences of 0's and 1's. These sequences code naturally into blocks, so the base set is $A = [Z/(2)]^m$. Since the image set must be properly larger than the base set, we take the image set to be $[Z/(2)]^n$, where $n>m$. A one-to-one map Φ from $[Z/(2)]^n$ is called an (m,n) encoding. An (m,n) encoding can detect errors in transmission if it takes many errors to change one encoded block into another encoded block. Thus two encoded words should be far apart in terms of numbers of errors required to transform one encoded word into another.

The weight of a vector x in $[Z/(2)]^n$ is defined to be the number of nonzero components of the vector and is denoted by $w(\mathbf{x})$. If **x** and **y** are two vectors in $[Z/(2)]^n$, then the Hamming distance between **x** and **y** is defined to be the number of nonzero components in **x**—**y** and is denoted by $d(\mathbf{x},\mathbf{y})$. Since addition in Z/(2) is the same as subtraction, and $1+1 = 0$, the distance from **x** to **y** is simply the number of components where the entry in **x** is different from the corresponding entry in **y**. It is easy to check that the Hamming distance is actually a metric, since $d(\mathbf{x},\mathbf{y})\geq 0$ for all vectors **x** and **y**,

$d(x,y) = 0$ if and only if $x = y$, and $d(x,y) + d(y,z) \geq d(x,z)$ for all vectors **x**, **y**, and **z** in $[Z/(2)]^n$.

This notion of distance determines the error-detecting capability of an encoding. An (m,n) encoding Φ is said to detect k or fewer errors if there is a procedure that indicates that a received block is incorrect whenever there are j errors, $1 \leq j \leq k$, in that received block.

Theorem 1 :- An (m,n) encoding Φ detects K or fewer errors if the minimal distance between encoded words is atleast K+1

Proof:- Assume that
Φ detects all sets of K of fewer errors.

Suppose x is an encoded word of distance j from another encoded word y.

$d(x,y) = j$; $1 \leq j \leq k$

If the encoded word x is transmitted and errors occur in exactly the components of x that differ from the corresponding components of y, then the received message will appear to be a Normal transmission of y.

Thus no error is noticed and yet j errors occurred in the transmission. This is a contradiction. Hence the minimum distance between encoded words must be atleast K+1.

Assume that the minimum distance between encoded words is atleast K+1. Thus, if j errors occur during the transmission of an encoded word x, then the received word will not be an encoded word. Hence there must have been errors in the transmission. Since the list of encoded words is finite, the received word can be compared with

each encoded word in the list to determine whether or not it represents an encoded vector.

If not, then errors have been made.

The above procedure is very inefficient, since each received word must essentially be compared with all possible encoded words.

An (m,n) emcoding Φ is said to correct K or fewer errors if there is a procedure that will correct a received word to yield the original encoded word, whenever j errors, $1 \leq j \leq k$, occur in transmission of the block.

Theorem 2:- An (m,n) encoding Φ corrects K or fewer errors if and only if the minimum distance between encoded words is atleast 2K+1

Proof:- Suppose that Φ corrects K or fewer errors x is an encoded word of distance $1 \leq j \leq k$ from the encoded word y.

Thus x-y = z , where the number of non-zero components in z is h.

Write z = a+b where a has atmost K nonzero entries and b has atmost K nonzero entries . If the transmission of the encoded word x results in x+a being received, then atmost K errors occurred during transmission. Likewise, if the transmission of the encoded word y results in y+b being received, then atmost K errors were made during transmission. However, x+a = y+b so the correct decoding is not possible which contradicts the ability of Φ to correct K or fewer errors. Hence the minimum distance between encoded words is atleast 2k−1

Assume minimum distance between encoded words is atleast 2k+1.If the transmission of an encoded word x results in an encoded word x+e, where $d(x, x+e) \leq k$. Then the encoded word x can be recovered, since x is the closest encoded word to x+e. Suppose y is atleast as close to x+e as x,

Then $d(x,y) \leq d(x+e, y) + d(x, x+e) \leq k+k=2k$

Which contradicts the fact that minimum distance between encoded words is atleast 2k+1. Thus to correct k or fewer errors, calculate the distance between the received word and each encoded word. Then the received word is decoded as the closest encoded word.

A common and simple example of a single-error detecting code is the parity-check code. The block $(a_1,\ldots a_m)$ is encoded as $(a_1,\ldots a_m, \Sigma a_i)$. This is an (m,m+1) encoding and has a minimum distance of 2 between encoded words. This code has the additional advantage that a received word does not have to be compared with all possible encoded words in order to detect an error. To see if an error occurred during transmission simply sum the components of the received vector modulo 2. If no errors occurred during transmission, then the sum of the components is $\Sigma a_i + \Sigma a_i = 0$ and if a single error occurred during transmission, then the sum of the components is 1.

Thus a vector in $[Z/(2)]^m$ can be encoded as that vector multiplied by a fixed m x n matrix. The encoding is one-to-one if and only if the rank of the matrix equals m, which automatically implies that $m \leq n$. Error detection capabilities are possible only if the vector space is mapped properly into the image space, hence in this case m is strictly less than n.