Appendix 1
Backus-Naur Form

BNF, or Backus-Naur form, is a notation for describing (part of) the syntax of "sentences" of a language. It was proposed in about 1959 for describing the syntax of Algol 60 by John Backus, one of the thirteen people on the Algol 60 committee. (John Backus, of IBM, was also one of the major figures responsible for FORTRAN.) Because of his modifications and extensive use of BNF as editor of the Algol 60 report, Peter Naur (University of Copenhagen) is also associated with it. The ideas were independently discovered earlier by Noam Chomsky, a linguist, in 1956. BNF and its extensions have become standard tools for describing the syntax of programming notations, and in many cases parts of compilers are generated automatically from a BNF description.

We will introduce BNF by using it to describe digits, integer constants, and simplified arithmetic expressions.

In BNF, the fact that 1 is a digit is expressed by

(A1.1) <digit> ::= 1

The term <digit> is delimited by angular brackets to help indicate that it cannot appear in "sentences" of the language being described, but is used only to help *describe* sentences. It is a "syntactic entity", much like "verb" or "noun phrase" in English. It is usually called a *nonterminal*, or *nonterminal symbol.* The symbol 1, on the other hand, can appear in sentences of the language being described, and is called a *terminal.*

(A1.1) is called a *production*, or (rewriting) *rule.* Its left part (the symbol to the left of ::=) is a nonterminal; its right part (to the right of ::=) is a nonempty, finite sequence of nonterminals and terminals. The symbol ::= is to be read as "may be composed of", so that (A1.1) can be read as

A <digit> may be composed of the sequence of symbols: 1

Two rules can be used to indicate a <digit> may be a 0 or a 1:

<digit> ::= 0 (a <digit> may be composed of 0)
<digit> ::= 1 (a <digit> may be composed of 1)

These two rules, which express different forms for the same nonterminal, can be abbreviated using the symbol |, read as "or", as

<digit> ::= 0 | 1 (A <digit> may be composed of 0 or 1)

This abbreviation can be used in specifying all the digits:

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

An integer constant is a (finite) sequence of one or more digits. Using the nonterminal <constant> to represent the class of integer constants, integer constants are defined recursively as follows:

(A1.2) <constant> ::= <digit>
<constant> ::= <constant> <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The first rule of (A1.2) is read as follows: a <constant> may be composed of a <digit>. The second rule is read as follows: a <constant> may be composed of another <constant> followed by a <digit>.

The rules listed in (A1.2) form a *grammar* for the language of <constant>s; the *sentences* of the language are the sequences of terminals that can be *derived* from the nonterminal <constant>, where sequences are derived as follows. Begin with the sequence of symbols consisting only of the nonterminal <constant>, and successively rewrite one of the nonterminals in the sequence by a corresponding right part of a rule, until the sequence contains only terminals.

Indicating the rewriting action by =>, the sentence 325 is derived:

(A1.3) <constant> => <constant> <digit> (rewrite using second rule)
=> <constant> 5 (rewrite <digit> as 5)
=> <constant> <digit> 5 (rewrite using second rule)
=> <constant> 2 5 (rewrite <digit> as 2)
=> <digit> 2 5 (rewrite using first rule)
=> 3 2 5 (rewrite <digit> as 3)

The derivation of one sequence of symbols from another can be defined schematically as follows. Suppose $U ::= u$ is a rule of the grammar, where U is a nonterminal and u a sequence of symbols. Then, for any

(possibly empty) sequences x and y define

$$x\ U\ y \Rightarrow x\ u\ y$$

The symbol => denotes a single derivation —one rewriting action. The symbol =>* denotes a sequence of zero or more single derivations. Thus,

<constant> 1 =>* <constant> 1

since <constant> 1 can be derived from itself in zero derivations. Also

<constant> 1 =>* <constant> <digit> 1

because of the second rule of grammar (A1.2). Finally,

<constant> 1 =>* 3 2 5 1

since (A1.3) showed that <constant> =>* 3 2 5.

A grammar for (simplified) arithmetic expressions

Now consider writing a grammar for arithmetic expressions that use addition, binary subtraction, multiplication, parenthesized expressions, and integer constants as operands. This is fairly easy to do:

<expr> ::= <expr> + <expr>
<expr> ::= <expr> − <expr>
<expr> ::= <expr> * <expr>
<expr> ::= (<expr>)
<expr> ::= <constant>

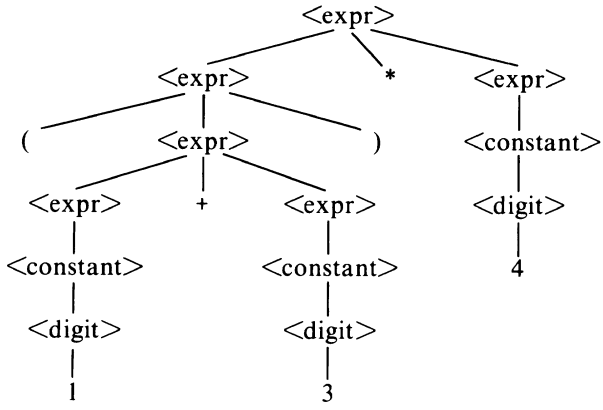
where <constant> is as described above in (A1.2). Here is a derivation of the expression (1+3)*4 according this grammar:

(A1.4) <expr> => <expr> * <expr>
=> (<expr>) * <expr>
=> (<expr> + <expr>) * <expr>
=> (<constant> + <expr>) * <expr>
=> (<constant> + <constant>) * <expr>
=> (<constant> + <constant>) * <constant>
=> (<digit> + <constant>) * <constant>
=> (<digit> + <digit>) * <constant>
=> (<digit> + <digit>) * <digit>
=> (1 + <digit>) * <digit>
=> (1 + 3) * <digit>
=> (1 + 3) * 4

Hence, <expr> =>* (1 + 3) + 4.

Syntax trees and ambiguity

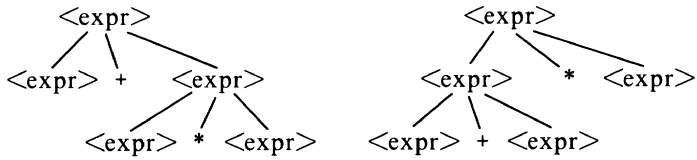
A sequence of derivations can be described by a *syntax tree.* As an example, the syntax tree for derivation (A1.4) is



In the syntax tree, a single derivation using the rule $U ::= u$ is expressed by a node U with lines emanating down to the symbols of the sequence u. Thus, for every single derivation in the sequence of derivations there is a nonterminal in the tree, with the symbols that replace it underneath. For example, the first derivation is <expr> => <expr> * <expr>, so at the top of the diagram above is the node <expr> and this node has lines emanating downward from it to <expr>, * and <expr>. Also, there is a derivation using the rule <digit> ::= 1, so there is a corresponding branch from <digit> to 1 in the tree.

The main difference between a derivation and its syntax tree is that the syntax tree does not specify the order in which some of the derivations were made. For example, in the tree given above it cannot be determined whether the rule <digit> ::= 1 was used before or after the rule <digit> ::= 3. To every derivation there corresponds a syntax tree, but more than one derivation can correspond to the same tree. These derivations are considered to be equivalent.

Now consider the set of derivations expressed by <expr> =>* <expr> + <expr> * <expr>. There are actually two different derivation trees for the two derivations of <expr> + <expr> * <expr>:

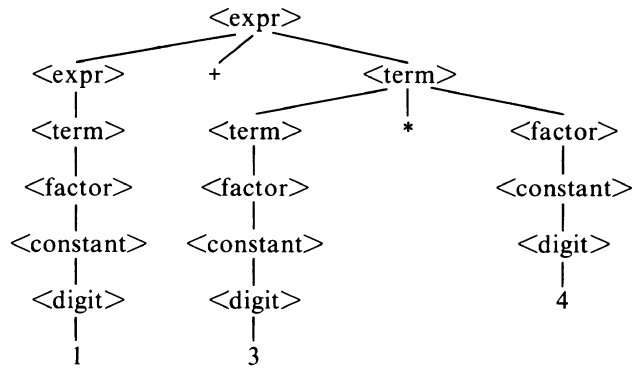


A grammar that allows more than one syntax tree for some sentence is called *ambiguous*. This is because the existence of two syntax trees allows us to "parse" the sentence in two different ways, and hence to perhaps give two meanings to it. In this case, the ambiguity shows that the grammar does not indicate whether + should be performed before or after *. The syntax tree to the left (above) indicates that * should be performed first, because the <expr> from which it is derived is in a sense an operand of the addition operator +. On the other hand, the syntax tree to the right indicates that + should be performed first.

One can write an unambiguous grammar that indicates that multiplication has precedence over plus (except when parentheses are used to override the precedence). To do this requires introducing new nonterminal symbols, <term> and <factor>:

```
<expr>     ::= <term> | <expr> + <term>
             | <expr> - <term>
<term>     ::= <factor> | <term> * <factor>
<factor>   ::= <constant> | ( <expr> )
<constant>::= <digit>
<constant>::= <constant> <digit>
<digit>    ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

In this gramar, each sentence has one syntax tree, so there is no ambiguity. For example, the sentence 1+3*4 has one syntax tree:



This syntax tree indicates that multiplication should be performed first, and, in general, in this grammar * has precedence over + except when the precedence is overridden by the use of parentheses.

Extensions to BNF

A few extension to BNF are used to make it easier to read and understand. One of the most important is the use of braces to indicate repetition: $\{x\}$ denotes zero or more occurrences of the sequence of symbols x. Using this extension, we can describe *<constant>* using one rule as

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

In fact, the grammar for arithmetic expressions can be rewritten as

```
<expr>     ::= <term> {+<term> | −<term>}
<term>     ::= <factor> {* <factor>}
<factor>   ::= <constant> | ( <expr> )
<constant>::= <digit> {<digit>}
<digit>    ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

References

The theory of syntax has been studied extensively. An excellent text on the material is *Introduction to Automata Theory, Languages and Computation* (Hopcroft, J.E. and J.D. Ullman; Addison-Wesley, 1979). The practical use of the theory in compiler construction is discussed in the texts *Compiler Construction for Digital Computers* (Gries, D.; John Wiley, 1971) and *Principles of Compiler Design* (Aho, A.V., and J. Ullman; Addison Wesley, 1977).

Appendix 2
Sets, Sequences, Integers, and Real Numbers

This appendix briefly defines the important types of variables used throughout the book. Sets will be described in more detail than the others, so that the reader can learn important material he might have missed earlier in his education.

Sets and operations on them

A set is a collection of distinct objects, or elements, as they are usually called. Because the word *collection* is just as vague as *set*, we give some examples to make the idea more concrete.

> The set $\{3, 5\}$ consists of the integers 3 and 5.
>
> The set $\{5, 3\}$ consists of the integers 3 and 5.
>
> The set $\{3, 3, 5\}$ consists of the integers 3 and 5.
>
> The set $\{3\}$ consists of the integer 3.
>
> The set $\{\}$ is called the *empty set*; it contains no elements. The Greek character ϕ is sometimes used to denote the empty set.

These examples illustrate one way of describing a set: write its elements as a list within braces { and }, with commas joining adjacent elements. The first two examples illustrate that the order of the elements in the list does not matter. The third example illustrates that an element listed more than once is considered to be in the set only once; elements of a set must be distinct. The final example illustrates that a set may contain zero elements, in which case it is called the empty set.

It is not possible to list all elements of an infinite set (a set with an infinite number of elements). In this case, one often uses dots to indicate that the reader should use his imagination, but in a conservative fashion,

in extending the list of elements actually given. For example,

$\{0, 1, 2, \cdots\}$ is the set of natural numbers

$\{\cdots, -2, -1, 0, 1, 2, \cdots\}$ is the set of all integers

$\{1, 2, 4, 8, 16, 32, \cdots\}$ is the set of powers of 2.

We can be more explicit using a different notation:

$$\{i \mid \textit{there is a natural number } j \textit{ satisfying } i = 2^j\}$$

In this notation, between { and | is an identifier i; between | and } is a true-false statement —a predicate. The set consists of all elements i that satisfy the true-false statement. In this case, the set consists of the powers of 2. The following describes the set of all even integers.

$$\{k \mid \textit{even}(k)\}$$

The notation can be extended somewhat:

$$\{(i, j) \mid i = j + 1\}$$

Assuming i and j are integer-valued, this describes the set of pairs

$$\{\cdots, (-1, -2), (0, -1), (1, 0), (2, 1), (3, 2), \cdots)$$

The *cardinality* or size of a set is the number of elements in it. The notations $|a|$ and $card(a)$ are often used to denote the cardinality of set a. Thus, $|\{\}| = 0$, $|\{1, 5\}| = 2$, and $card(\{3, 3, 3\}) = 1$.

The following three operations are used build new sets: set union $\cup$, set intersection $\cap$ and set difference $-$.

$a \cup b$	is the set consisting of elements that are in at least one of the sets a and b
$a \cap b$	is the set consisting of elements that are in both a and b
$a - b$	is the set consisting of elements that are in a but not in b

For example, if $a = \{A, B, C\}$ and $b = \{B, C, D\}$, then $a \cup b = \{A, B, C, D\}$, $a \cap b = \{B, C\}$ and $a - b = \{A\}$.

Besides tests for set equality or inequality (e.g. $a = b$, $a \neq \{2, 3, 5\}$), the following three operations yield a Boolean value T or F (true or false):

$x \in a$	has the value of "x is a member of set a"
$x \notin a$	is equivalent to $\neg(x \in a)$
$a \subset b$	has the value of "set a is a subset of set b" —i.e. each element of a is in b

Thus, $1 \in \{2, 3, 5, 7\}$ is false, $1 \notin \{2, 3, 5, 7\}$ is true, $\{1, 3, 5\} \subset \{1, 3, 5\}$ is true, $\{3, 1, 5\} \subset \{1, 3, 5, 0\}$ is true, and $\{3, 1, 5, 2\} \subset \{1, 3, 5, 0\}$ is false.

It is advantageous from time to time to speak of the minimum and maximum values in a set (if an ordering is defined on its elements): $min(a)$ denotes the minimum value in set a and $max(a)$ the maximum value.

Finally, we describe a command that is useful when programming using sets. Let a be a nonempty set and x be a variable that can contain values of the type of the set elements. Execution of the command

$$Choose(a, x)$$

stores in x one element of set a. Set a remains unchanged. This command is nondeterministic (see chapter 7), because it is not known before its execution which element of a will be stored in x. The command is assumed to be used only with finite sets. Its use with infinite sets causes problems, which are beyond the scope of this book (see [16] for a discussion of unbounded and bounded nondeterminism). Further, in our programs the sets we deal with are finite.

$Choose(a, x)$ is defined in terms of weakest preconditions as follows (see chapter 7):

$$wp(Choose(a, x), R) \;=\; a \neq \{\} \wedge (\boldsymbol{A}\, i\colon i \in a\colon R_i^x)$$

Sequences

A *sequence* is a list of elements (joined by commas and delimited by parentheses). For example, the sequence (1, 3, 5, 3) consists of the four elements 1, 3 , 5 , 3, in that order, and () denotes the empty sequence. As opposed to sets, the ordering of the elements in a sequence is important.

The length of a sequence s, written $|s|$, is the number of elements in it.

Catenation of sequences with sequences and/or values is denoted by $|$. Thus,

$$\begin{aligned}(1, 3, 5) \mid (2, 8) &= (1, 3, 5, 2, 8)\\ (1, 3, 5) \mid 8 \mid 2 &= (1, 3, 5, 8, 2)\\ (1, 3, 5) \mid () &= (1, 3, 5)\end{aligned}$$

In programming, the following notation is used to refer to elements of a sequence. Let variable s be a sequence with n elements. Then

$$s = (s[0], s[1], s[2], \cdots, s[n-1])$$

That is, s[0] refers to the first element, $s[1]$ to the second, and so forth. Further, the notation $s[k..]$, where $0 \leqslant k \leqslant n$, denotes the sequence

$$s[k..] = (s[k], s[k+1], \cdots, s[n-1])$$

That is, $s[k..]$ denotes a new sequence that is the same as s but with the first k elements removed. For example, if s is not empty, the assignment

$$s := s[1..]$$

deletes the first element of s. Executing the assignment when $s = ()$ causes abortion, because the expression $s[1..]$ is not defined in that case.

One can implement a last-in-first-out stack using a sequence s by limiting the operations on s to

$s[0]$	reference the top element of the stack
$s := ()$	empty the stack
$x, s := s[0], s[1..]$	Pop an element into variable x
$s := v \mid s$	Push value v onto the stack

One can implement a (first-in-first-out) queue using a sequence s by limiting the operations on s to

$s[0]$	Reference the front element of the queue
$s := ()$	empty the queue
$x, s := s[0], s[1..]$	Delete the front element and store it in x
$s := s \mid v$	Insert value v at the rear

Using the sequence notation, rather than the usual pop and push of stacks and insert into and delete from queues, may lead to more understandable programs. The notion of assignment is already well understood —see chapter 9— and is easy to use in this context.

Operations on integers and real numbers

We typically use the following sets:

The set of integers: $\{\cdots, -2, -1, 0, 1, 2, \cdots\}$
The set of natural numbers: $\{0, 1, 2, \cdots\}$

We also use the set of real numbers, although on any machine this set and operations on it are approximated by some form of floating point numbers and operations. Nevertheless, we assume that real arithmetic is performed, so that problems with floating point are eliminated.

The following operations take as operands either integers or real numbers:

$+, -, *$	addition, subtraction, multiplication
$/$	division x / y; yields a real number
$<, \leqslant, =, \geqslant, >, \neq$	the relational operators: $x < y$ is read "x is less than y" $x \leqslant y$ is read "x is at most y" $x = y$ is read "x equals y" $x \geqslant y$ is read "x is at least y" $x > y$ is read "x exceeds y" $x \neq y$ is read "x differs from y"
$abs(x)$ or $\lvert x \rvert$	absolute value of x: **if** $x < 0$ **then** $-x$ **else** x
$floor(x)$	greatest integer not more than x
$ceil(x)$	smallest integer not less than x
$min(x, y, \cdots)$	The minimum of $x, y, \cdots$. The minimum of an empty set is ∞ (infinity)
$max(x, y, \cdots)$	The maximum of $x, y, \cdots$. The maximum of an empty set is $-\infty$ (infinity)
$log(x)$	base 2 logarithm of x: $y = log(x)$ *iff* $x = 2^y$

The following operations take only integers as operands.

$\div$	$x \div y$ is the greatest integer at most x / y
x **mod** y	the remainder when x is divided by y (for $x \geqslant 0, y > 0$)
$even(x)$	"x is an even integer", or x **mod** $2 = 0$
$odd(x)$	"x is an odd integer", or x **mod** $2 = 1$

Appendix 3
Relations and Functions

Relations

Let A and B be two sets. The *Cartesian product* of A and B, written $A \times B$, is the set of ordered pairs (a, b) where a is in A and b is in B:

$$A \times B = \{(a, b) \mid a \in A \wedge b \in B\}$$

The Cartesian product is named after the father of analytic geometry, Rene Descartes, a 17th century mathematician and philosopher. The number of elements in $A \times B$ is $|A| * |B|$; hence the name Cartesian *product*.

A *binary relation* over the sets A and B is a subset of $A \times B$. Since we will be dealing mainly with binary relations, we drop the adjective *binary* and call them simply *relations*. A few words will be said about other relations at the end of this Appendix.

Let P be the set of people. One relation over $P \times P$ is the relation *parent*:

$$parent = \{(a, b) \mid b \text{ is } a\text{'s parent}\}$$

Let N be the set of integers. One relation over $N \times N$ is the *successor* relation:

$$succ = \{(i, i+1) \mid i \in N\}$$

The following relation associates with each person the year in which he left his body:

$$died\_in = \{(p, i) \mid \text{person } p \text{ died in year } i\}$$

The *identity relation* over $A \times A$, denoted by I, is the relation

$$I = \{(a, a) \mid a \in A\}$$

When dealing with binary relations, we often use the name of a relation as a binary operator and use infix notation to indicate that a pair belongs in the relation. For example, we have

$$c \ \textit{parent} \ d \ \ \textit{iff} \ \ (d, c) \in \{(a, b) \mid b \ \textit{is a's parent}\}$$
$$i \ \textit{succ} \ j \ \ \textit{iff} \ \ i+1 = j$$
$$q \ \textit{died\_in} \ j \ \ \textit{iff} \ \ (q, j) \in \{(p, i) \mid \textit{person } p \ \textit{died in year } i\}$$

From the three relations given thus far, we can conclude several things. For any value a there may be different pairs (a, b) in a relation. Such a relation is called a *one-to-many* relation. Relation *parent* is one-to-many, because most people have more than one parent.

For any value b there may be different pairs (a, b) in a relation. Such a relation is called a *many-to-one* relation. Many people may have died in any year, so that for each integer i there may be many pairs (p, i) in relation *died_in*. But for any person p there is at most one pair (p, i) in *died_in*. Relation *died_in* is an example of a *many-to-one* relation.

In relation *succ*, no two pairs have the same first value and no two pairs have the same second value. Relation *succ* is an example of a *one-to-one* relation.

A relation on $A \times B$ may contain no pair (a, b) for some a in A. Such a relation is called a *partial* relation. On the other hand, a relation on $A \times B$ is *total* if for each $a \in A$ there exists a pair (a, b) in the relation. Relation *died_in* is partial, since not all people have died yet. Relation *succ* is total (on $N \times N$).

If relation R on $A \times B$ contains a pair (a, b) for each b in B, we say that R is *onto* B. Relation *parent* is onto, since each child has a parent (assuming there was no beginning).

Let R and S be two relations. Then the *composition* $R \circ S$ of R and S is the relation defined by

$$a \ R \circ S \ c \ \ \textit{iff} \ \ (E b : a \ R \ b \wedge b \ S \ c)$$

For example, the relation *parent* $\circ$ *parent* is the relation *grandparent*. Rrelation *died_in* $\circ$ *succ* associates with each person the year after the one in which the person died.

Composition is *associative*. This means the following. Let R, S and T be three relations. Then $(R \circ S) \circ T = R \circ (S \circ T)$. This fact is easily deduced from the definitions of relation and composition. Because com-

position is associative, we usually omit the parentheses and write simply $R \circ S \circ T$.

The composition of a relation with itself is denoted with a superscript 2:

$$\begin{array}{ll} parent^2 & \textit{is equivalent to } parent \circ parent \\ succ^2 & \textit{is equivalent to } succ \circ succ \end{array}$$

Similarly, for any relation R and natural number i one defines R^i as

$$\begin{array}{lll} R^0 & = & I, \qquad \text{the identity relation} \\ R^i & = & R \circ R^{i-1}, \quad \text{for } i > 0 \end{array}$$

For example,

$$\begin{array}{lll} parent^0 & = & I \\ parent^1 & = & parent \\ parent^2 & = & grandparent \\ parent^3 & = & great\text{-}grandparent \end{array}$$

and

$$(i \; succ^k \; j) \quad \textit{iff} \quad i+k=j$$

Looking upon relations as sets and using the superscript notation, we can define the *closure* R^+ and *transitive closure* R^* of a relation R as follows.

$$\begin{array}{lll} R^+ & = & R^1 \cup R^2 \cup R^3 \cup \cdots \\ R^* & = & R^0 \cup R^1 \cup R^2 \cup \cdots \end{array}$$

In other words, a pair (a, b) is in R^+ if and only if it is in R^i for some $i > 0$. Here are some examples.

$parent^+$ is the relation *ancestor*
$i \; succ^+ \; j$ *iff* $i+k=j$ *for some* $k>0$, *i.e.* $i<j$
$i \; succ^* \; j$ *iff* $i \leqslant j$

Finally, we can define the inverse R^{-1} of a relation R:

$$b \; R^{-1} \; a \quad \textit{iff} \quad a \; R \; b$$

That is, (b, a) is in the inverse R^{-1} of R if and only if (a, b) is in R. The inverse of *parent* is *child*, the inverse of $<$ is $>$, the inverse of $\leqslant$ is $\geqslant$, and the inverse of the identity relation I is I itself.

Functions

Let A and B be sets. A function f from A to B, denoted by

$$f: A \rightarrow B$$

is a relation that for each element a of A contains at most one pair (a, b) —i.e. is a relation that is not one-to-many. The relation *parent* is not a function, because a child can have more than one parent —for each person p there may be more than one pair (p, q) in the set *parent*. The relations *succ* and *died_in* are functions.

We write each pair in a function f as $(a, f(a))$. The second value, $f(a)$, is called the value of function f at the argument a. For example, for the function *succ* we have

$$succ(i) = i+1, \text{ for all natural numbers } i,$$

because *succ* is the set of pairs

$$\{(..., (-2, -1), (-1, 0), (0, 1), (1, 2), ...)\}$$

Note carefully the three ways in which a function name f is used. First, f denotes a set of pairs such that for any value a there is at most one pair (a, b). Second, $a\ f\ b$ holds if (a, b) is in f. Third, $f(a)$ is the value associated with a, that is, $(a, f(a))$ is in the function (relation) f.

The beauty of defining a function as a restricted form of relation is that the terminology and theory for relations carries over to functions. Thus, we know what a one-to-one function is. We know that composition of (binary) functions is associative. We know, for any function, what f^0, f^1, f^2, f^+ and f^* mean. We know what the inverse f^{-1} of f is. We know that f^{-1} is a function *iff* f is not many-to-one.

Functions from expressions to expressions

Consider an expression, for example $x*y$. We can consider $x*y$ as a function of one (or more) of its identifiers, say y:

$$f(y) = x*y$$

In this case, we can consider f to be a function from expressions to expressions. For example,

$$f(2) = x*2$$
$$f(x+2) = x*(x+2)$$
$$f(x*2) = x*x*2$$

Thus, to apply the function to an argument means to replace *textually* the identifier y everywhere within the expression by the argument. In making the replacement, one should insert parentheses around the argument to maintain the precedence of operators in the expression (as in the second example), but we often leave them out where it doesn't matter. This textual substitution is discussed in more detail in section 4.4. The reason for including it here is that the concept is used earlier in the book, and the reader should be familiar with it.

n-ary relations and functions

Thus far, we have dealt only with binary relations. Suppose we have sets A_0, ..., A_n for $n > 0$. Then one can define a relation on $A_0 \times A_2 \times \cdots \times A_n$ to be a set of ordered tuples

$$(a_0, a_2, \cdots, a_n)$$

where each a_i is a member of set A_i.

Suppose g is such a relation. Suppose, further, that for each distinct $n-1$ tuple $(a_0, \cdots, a_{n-1})$ there is at most one n-tuple $(a_0, \ldots, a_{n-1}, a_n)$ in g. Then g is an n-ary function —a function of n arguments— where the value a_n in each tuple is the value of the function when applied to the arguments consisting of the first $n-1$ values:

$$(a_0, a_2, \cdots, a_{n-1}, g(a_0, \cdots, a_{n-1})) \quad \text{and}$$
$$g(a_0, \cdots, a_{n-1}) = a_n$$

The terminology used for binary relations and functions extends easily to n-ary relations and functions.

Appendix 4
Asymptotic Execution Time Properties

Execution time measures of algorithms that hold for any implementation of the algorithms and for any computer, especially for large input values (e.g. large arrays), are useful. The rest of this section is devoted to describing a suitable method for measuring execution times in this sense. The purpose is not to give an extensive discussion, but only to outline the important ideas for those not already familiar with them.

First, assign units of execution time to each command of a program as follows. The assignment command and *skip* are each counted as 1 unit of time, since their executions take essentially the same time whenever they are executed. An alternative command is counted as the maximum number of units of its alternatives (this may be made finer in some instances). An iterative command is counted as the sum of the units for each of its iterations, or as the number of iterations times the maximum number of units required by each iteration.

One can understand the importance of bound functions for loops in estimating execution time as follows. If a program has no loops, then its execution time is bounded no matter what the input data is. Only if it has a loop can the time depend on the input values, and then the number of loop iterations, for which the bound function gives an upper bound, can be used to give an estimate on the time units used.

Consider the three following loops, $n \geqslant 0$:

$i := n$; **do** $i > 1 \rightarrow i := i - 1$ **od**

$i := n$; $j := 0$; **do** $i > 1 \rightarrow i := i - 1$; $j := 0$ **od**

$i := n$; **do** $i > 1 \rightarrow i := i \div 2$ **od**

The first requires n units of time; the second $2n$. The units of time required by the third program is more difficult to determine. Suppose n

is a power of 2, so that

$$i = 2^k$$

holds for some k. Then dividing i by 2 is equivalent to subtracting 1 from k: $(2^k) \div 2 = 2^{k-1}$. Therefore, each iteration decreases k by 1. Upon termination, $i = 1 = 2^0$, so that $k = 0$. Hence, a suitable bound function is $t = k$, and, in fact, the loop iterates *exactly* k times. The third program makes exactly $ceil(\log n)$ iterations, for any $n > 0$.

Whenever an algorithm iteratively divides a variable by 2, you can be sure that a log factor is creeping into its execution time. We see this in Binary Search (exercise 4 of section 16.3) and in Quicksort (18.2). This log factor is important, because $\log n$ is ever so much smaller than n, as n grows large.

The execution time of the first and second programs given above can be considered roughly the same, while that of the third is much lower. This is illustrated in the following table, which gives values of n, $2n$ and $\log n$ for various values of n. Thus, for $n = 32768$ the first program requires 32769 basic units of time, the second twice as many, and the third only 15!

n:	1	2	64	128	32768
$2n$:	2	4	128	256	65536
$\log n$:	0	1	6	7	15

We need a measure that allows us to say that the third program is by far the fastest and that the other two are essentially the same. To do this, we define the *order of execution time.*

(A4.1) **Definition.** Let $f(n)$ and $g(n)$ be two functions. We say that $f(n)$ is (no more than) order $g(n)$, written $O(g(n))$, if a constant $c > 0$ exists such that, for all (except a possibly finite number) positive values of n,

$$f(n) \leqslant c^* g(n).$$

Further, $f(n)$ *is proportional to* $g(n)$ if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$. We also say that $f(n)$ and $g(n)$ are *of the same order.* □

Example 1. Let $f(n) = n+5$ and $g(n) = n$. Choose $c = 5$. Then, for $1 < n$, $f(n) = n+5 \leqslant 5n = 5^* g(n)$. Hence, $f(n)$ is $O(g(n))$. Choosing $c = 1$ indicates that $g(n)$ is $O(f(n))$. Hence $n+5$ is proportional to n. □

Example 2. Let $f(n)=n$ and $g(n)=2n$. With $c=1$ we see that $f(n)$ is $O(g(n))$. With $c=1/2$, we see that $g(n)$ is $O(f(n))$. Hence n is proportional to $2n$. For any constants $K1\neq 0$ and $K2$, $K1*n+K2$ is proportional to n. □

Since the first and second programs given above are executed in n and $2n$ units, respectively, their execution times are of the same order. Secondly, one can prove that $\log n$ is $O(n)$, but *not* vice versa. Hence the order of execution time of the third is less than that of the first two programs.

We give below a table of typical execution time orders that arise frequently in programming, from smallest to largest, along with frequent terms used for them. They are given in terms of a single input parameter n. In addition, the (rounded) values of the orders are given for $n=100$ and $n=1000$, so that the difference between them can be seen.

Order	$n=100$	$n=1000$	Term Used
1	1	1	Constant-time algorithm
$\log n$	7	10	logarithmic algorithm
$\sqrt{n}$	10	32	
n	100	1000	linear algorithm
$n \log n$	700	10000	
$n\sqrt{n}$	1000	31623	
n^2	10000	1000000	quadratic (or simply n^2) algorithm
n^3	1000000	10^9	cubic algorithm
2^n	$1.26*10^{30}$	$\sim 10^{300}$	exponential algorithm

For algorithms that have several input values the calculation of the order of execution time becomes more difficult, but the technique remains the same. When comparing two algorithms, one should first compare their execution time orders, and, if they are the same, then proceed to look for finer detail such as the number of times units required, number of array comparisons made, etc.

An algorithm may require different times depending on the configuration of the input values. For example, one array $b[1{:}n]$ may be sorted in n steps, another array $b'[1{:}n]$ in n^2 steps by the same algorithm. In this case there are two methods of comparing the algorithms: average- or expected-case time analysis and worst-case time analysis. The former is quite difficult to do; the latter usually much simpler.

As an example, Linear Search, (16.2.5), requires n time units in the worst case and $n/2$ time units in the average case, if one assumes the value being looked for can be in any position with equal probability.

Answers to Exercises

Answers for Chapter 1

1. We show the evaluation of the expressions for state $s1$.

(a) $\neg(m \vee n) = \neg(T \vee F) = \neg T = F$

(b) $\neg m \vee n = \neg T \vee F = F \vee F = F$

(c) $\neg(m \wedge n) = \neg(T \wedge F) = \neg F = T$

(d) $\neg m \wedge n = \neg T \wedge F = F \wedge F = F$

(e) $(m \vee n) \Rightarrow p = (T \vee F) \Rightarrow T = T \Rightarrow T = T$

(f) $m \vee (n \Rightarrow p) = T \vee (F \Rightarrow T) = T \vee T = T$

(g) $(m = n) \wedge (p = q) = (F = F) \wedge (T = F) = T \wedge F = F$

(h) $m = (n \wedge (p = q)) = F = (F \wedge (T = F))$
$= F = (F \wedge F) = F = F = T$

(i) $m = (n \wedge p = q) = F = (F \wedge T = F) = F = (F = F) = F = T = F$

(j) $(m = n) \wedge (p \Rightarrow q) = (F = T) \wedge (F \Rightarrow T) = F \wedge T = F$

(k) $(m = n \wedge p) \Rightarrow q = (F = T \wedge F) \Rightarrow T = (F = F) \Rightarrow T = T \Rightarrow T = T$

(l) $(m \Rightarrow n) \Rightarrow (p \Rightarrow q) = (F \Rightarrow F) \Rightarrow (F \Rightarrow F) = T \Rightarrow T = T$

(m) $(m \Rightarrow (n \Rightarrow p)) \Rightarrow q = (F \Rightarrow (F \Rightarrow F)) \Rightarrow F$
$= (F \Rightarrow T) \Rightarrow F = T \Rightarrow F = F$

2. a,b

b	c	d	$b \vee c$	$b \vee c \vee d$	$b \wedge c$	$b \wedge c \wedge d$
T	T	T	T	T	T	T
T	T	F	T	T	T	F
T	F	T	T	T	F	F
T	F	F	T	T	F	F
F	T	T	T	T	F	F
F	T	F	T	T	F	F
F	F	T	F	T	F	F
F	F	F	F	F	F	F

3. Let the Boolean identifiers and their meanings be:

xlessy: $x < y$ *xequaly*: $x = y$ *xgreatery*: $x > y$ *xatleasty*: $x \geqslant y$
ylessz: $y < z$ *yequalz*: $y = z$ *ygreaterz*: $y > z$
vequalw: $v = w$
beginxlessy: Execution of program P begins with $x < y$
*beginxless*0: Execution of program P begins with $x < 0$
*endyequal*2*powerx*: Execution of P terminates with $y = 2^x$
noend: Execution of program P does not terminate

We give one possible proposition; there are others.

(a) *xlessy* $\vee$ *xequaly*
(d) *xlessy* $\wedge$ *ylessz* $\wedge$ *vequalw*
(k) *beginxless*0 $\Rightarrow$ *noend*

Answers for Chapter 2

1. Truth table for the first Commutative law (only) and the law of Negation:

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

Truth table for the first Distributive law (only) (since the last two columns are the same, the two expressions heading the columns are equivalent and the law holds):

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

3.
$\neg T = \neg(T \vee \neg T)$ (Excluded Middle)
$= \neg T \wedge \neg\neg T$ (De Morgan)
$= \neg T \wedge T$ (Negation)
$= T \wedge \neg T$ (Commutativity)
$= F$ (Contradiction)

5. Column 1: (b) Contradiction, (c) **or**-simpl., (d) **or**-simpl., (e) **and**-simpl., (f) Identity, (g) Contradiction, (h) Associativity, (i) Distributivity, (j) Distributivity, (k) Commutativity (twice), (l) Negation, (m) De Morgan.

6. (a) $x \vee (y \vee x) \vee \neg y$
$= x \vee (x \vee y) \vee \neg y$ (Commut.)
$= (x \vee x) \vee (y \vee \neg y)$ (Ass.)
$= x \vee T$ (**or**-simpl., Excl. Middle)
$= T$ (**or**-simpl.)

(b) $(x \vee y) \wedge (x \vee \neg y)$
$= x \vee (y \wedge \neg y)$ (Dist.)
$= x \vee F$ (Contradiction)
$= x$ (**or**-simpl.)

(g) $\neg x \Rightarrow (x \wedge y)$
$= x \vee (x \wedge y)$ (Imp., Neg.)
$= x$ (**or**-simpl.)

(h) $T \Rightarrow (\neg x \Rightarrow x)$
$= \neg T \vee (x \vee x)$ (Imp., Neg.)
$= F \vee (x \vee x)$ (exercise 3)
$= x$ (**or**-simpl., twice)

7. Proposition e is transformed using the equivalence laws in 6 major steps:

1. Use the law of Equality to eliminate all occurrences of $=$.

2. Use the law of Implication to eliminate all occurrences of $\Rightarrow$.

3. Use De Morgan's laws and the law of Negation to "move **not** in" so that it is applied only to identifiers and constants. For example, transform $\neg (a \vee (F \wedge \neg c))$ as follows:

$$\begin{aligned} &\neg(a \vee (F \wedge \neg c)) \\ &= \neg a \wedge \neg(F \wedge \neg c) \\ &= \neg a \wedge (\neg F \vee \neg \neg c) \\ &= \neg a \wedge (\neg F \vee c) \end{aligned}$$

4. Use $\neg F = T$ and $\neg T = F$ to eliminate all occurrences of $\neg F$ and $\neg T$ (see exercises 3 and 4).

5. The proposition now has the form $e_0 \vee \cdots \vee e_n$ for some $n \geqslant 0$, where each of the e_i has the form $(g_0 \wedge \cdots \wedge g_m)$. Perform the following until all g_j in all e_i have one of the forms id, $\neg id$, T and F:

> Consider some e_i with a g_j that is not in the desired form. Use the law of Commutativity to place it as far right as possible, so that it becomes g_m. Now, g_m must have the form $(h_0 \vee \cdots \vee h_k)$, so that the complete e_i is
>
> $$g_0 \wedge \cdots \wedge g_{m-1} \wedge (h_0 \vee \cdots \vee h_k)$$
>
> Use Distributivity to replace this by

$$(g_0 \wedge \cdots \wedge g_{m-1} \wedge h_0) \vee \cdots \vee (g_0 \wedge \cdots \wedge g_{m-1} \wedge h_k)$$

This adds m propositions e_i at the main level, but reduces the level of nesting of operators in at least one place. Hence, after a number of iterations it must terminate.

6. The proposition now has the form $e_0 \vee \cdots \vee e_n$ for some $n \geqslant 0$, where each of the e_i has the form $(g_0 \wedge \cdots \wedge g_m)$ and the g_j are id, $\neg id$, T or F. Use the laws of Commutativity, Contradiction, Excluded Middle and **or**-simplification to get the proposition in final form. If any of the e_i reduces to T then reduce the complete proposition to T; if any reduces to F use the law of **or**-simplification to eliminate it (unless the whole proposition is F).

9. The laws have already been proved to be tautologies in exercise 1. We now show that use of the rule of Substitution generates a tautology, by induction on the form of proposition $E(p)$, where p is an identifier.

Case 1: $E(p)$ is either T, F or an identifier that is not p. In this case, $E(e1)$ and $E(e2)$ are both E itself. By the law of Identity, $E = E$, so that a tautology is generated.

Case 2: $E(p)$ is p. In this case, $E(e1)$ is $e1$ and $E(e2)$ is $e2$, and by hypothesis $e1 = e2$ is a tautology.

Case 3: $E(p)$ has the form $\neg E1(p)$. By induction, $E1(e1) = E1(e2)$. Hence, $E1(e1)$ and $E1(e2)$ have the same value in every state. The following truth table then establishes the desired result:

$E1(e1)$	$E1(e2)$	$\neg E1(e1)$	$\neg E1(e2)$	$\neg E1(e1) = \neg E1(e2)$
T	T	F	F	T
F	F	T	T	T

Case 4: $E(p)$ has the form $E1(p) \wedge E2(p)$. By induction, we have that $E1(e1) = E1(e2)$ and $E2(e1) = E2(e2)$. Hence, $E1(e1)$ and $E1(e2)$ have the same value in every state, and $E2(e1)$ and $E2(e2)$ have the same value in every state. The following truth table then establishes the desired result:

$E1(e1)$	$E1(e2)$	$E2(e1)$	$E2(e2)$	$E1(e1) \wedge E2(e1)$	$E1(e2) \wedge E2(e2)$
T	T	T	T	T	T
T	T	F	F	F	F
F	F	T	T	F	F
F	F	F	F	F	F

The rest of the cases, $E(p)$ having the forms $E1(p) \vee E2(p)$, $E1(p) \Rightarrow E2(p)$ and $E1(p) = E2(p)$ are similar and are not shown here.

We now show that use of the rule of Transitivity generates only tautologies. Since $e1 = e2$ and $e2 = e3$ are tautologies, we know that $e1$ and $e2$ have the same value in every state and that $e2$ and $e3$ have the same value in every state. The following truth table establishes the desired result:

$e1$	$e2$	$e3$	$e1 = e3$
T	T	T	T
F	F	F	T

10. Reduce e to a proposition $e1$ in conjunctive normal form (see exercise 8). By exercise 9, $e = e1$ is a tautology, and since e is assumed to be a tautology, $e1$ must be a tautology. Hence it is true in all states. Proposition $e1$ is T, is F or has the form $e_0 \wedge \cdots \wedge e_n$, where each of the e_i has the form $g_0 \vee \cdots \vee g_m$ and each g_i is id or $\neg id$ and the g_i are distinct. Proposition $e1$ cannot have the latter form, because it is not a tautology. It cannot be F, since F is not a tautology. Hence, it must be T, and $e = T$ has been proved.

Answers for Section 3.2

3. (a) **From** $p \wedge q, p \Rightarrow r$ **infer** r

1	$p \wedge q$	pr 1
2	$p \Rightarrow r$	pr 2
3	p	$\wedge$-E, 1
4	r	$\Rightarrow$-E, 2, 3

or **From** $p \wedge q, p \Rightarrow r$ **infer** p

1	p	$\wedge$-E, pr 1
2	p	$\Rightarrow$-E, pr 2, 1

3. (b) **From** $p = q$, q **infer** p

1	$p = q$	pr 1
2	q	pr 2
3	$q \Rightarrow p$	=-E, 1
4	p	$\Rightarrow$-E, 3, 2

3. (c) **From** $p, q \Rightarrow r, p \Rightarrow r$ **infer** $p \wedge r$

1	r	$\Rightarrow$-E, pr 3, pr 1
2	$p \wedge r$	$\wedge$-I, pr 1, 1

4. Proof of 3(a). Since $p \wedge q$ is true, p must be true. From $p \Rightarrow r$, we then conclude that r must be true.

Proof of 3(b). Since $p = q$ is true, q being true means that p must be true.

Answers for Section 3.3

1. Infer $(p \wedge q \wedge (p \Rightarrow r)) \Rightarrow (r \vee (q \Rightarrow r))$

1	$(p \wedge q \wedge (p \Rightarrow r)) \Rightarrow (r \vee (q \Rightarrow r))$	$\Rightarrow$-I, (3.2.11)

2. Infer $(p \wedge q) \Rightarrow (p \vee q)$

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

4. Infer $p = p \vee p$

1	**From** p **infer** $p \vee p$		
	1.1	$p \vee p$	$\vee$-I, pr 1
2	$p \Rightarrow p \vee p$		$\Rightarrow$-I, 1
3	**From** $p \vee p$ **infer** p		
	3.1	p	$\vee$-E, pr 1, (3.3.3), (3.3.3)
4	$p \vee p \Rightarrow p$		$\Rightarrow$-I, 3
5	$p = p \vee p$		=-I, 2, 4

8. The reference on line 2.2 to line 2 is invalid.

11. From $\neg q$ **infer** $q \Rightarrow p$

1	$\neg q$		pr 1
2	**From** q **infer** p		
	2.1	q	pr 1
	2.2	**From** $\neg p$ **infer** $q \wedge \neg q$	
		2.2.1 $q \wedge \neg q$	$\wedge$-I, 2.1, 1
	2.3	p	$\neg$-E, 2.2
2	$q \Rightarrow p$		$\Rightarrow$-I, 2

26. **From** p **infer** p

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

27. The following "English" proofs are not intended to be particularly noteworthy; they only show how one might attempt to argue in English. Building truth tables or using the equivalence transformation system of chapter 2 is more reasonable.

Many of these proofs rely on the property that a proposition $b \Rightarrow c$ is true in any state in which the consequent c is true, and hence to prove that $b \Rightarrow c$ is a tautology one need only investigate states in which c is false.

27. Proof of 1. The proposition $(p \wedge q \wedge (p \Rightarrow r)) \Rightarrow (r \vee (q \Rightarrow r))$ is true because it was already proven in (3.2.11) that $(r \vee (q \Rightarrow r))$ followed from $(p \wedge q \wedge (p \Rightarrow r))$.

27. Proof of 2. If $p \wedge q$ is true, then p is true. Hence, anything "ored" with p is true, so $p \vee q$ is true.

27. Proof of 3. If q is true, then so is $q \wedge q$.

27. Proof of 5. Suppose p is true. Then $e \Rightarrow p$ is true no matter what e is. Hence $(r \vee s) \Rightarrow p$ is true. Hence, $p \Rightarrow ((r \vee s) \Rightarrow p)$ is true.

Answers for Section 3.4

1. **From** $p \Rightarrow (q \Rightarrow (p \vee q))$ **infer** $(p \wedge q) \Rightarrow (p \vee q)$

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

8.(a) **From** $b \vee c$ **infer** $\neg b \Rightarrow c$

1	$b \vee c$	pr 1
2	**From** $\neg b$ **infer** c	
	2.1 $\quad c$	(3.4.6), 1, pr 1
3	$\neg b \Rightarrow c$	$\Rightarrow$-I, 2

8.(b) **From** $\neg b \Rightarrow c$ **infer** $b \vee c$

1	$\neg b \Rightarrow c$	pr 1
2	$b \vee \neg b$	(3.4.14)
3	**From** b **infer** $b \vee c$	
	3.1 | $b \vee c$	$\vee$-I, pr 1
4	$b \Rightarrow b \vee c$	$\Rightarrow$-I, 3
5	**From** $\neg b$ **infer** $b \vee c$	
	5.1 | c	$\Rightarrow$-E, 1, pr 1
	5.2 | $b \vee c$	$\vee$-I, 5.1
6	$\neg b \Rightarrow b \vee c$	$\Rightarrow$-I, 5
7	$b \vee c$	$\vee$-E, 2, 4, 6

8.(c) **Infer** $b \vee c = (\neg b \Rightarrow c)$

1	$(b \vee c) \Rightarrow (\neg b \Rightarrow c)$	$\Rightarrow$-I, 8(a)
2	$(\neg b \Rightarrow c) \Rightarrow b \vee c$	$\Rightarrow$-I, 8(b)
3	$b \vee c = (\neg b \Rightarrow c)$	=-I, 1, 2

10. We prove the theorem by induction on the structure of expression $E(p)$.

Case 1: E(p) is the single identifier p. In this case, $E(e1)$ is simply $e1$ and $E(e2)$ is $e2$, and we have the proof

From $e1 = e2$, $e1$ **infer** $e2$

1	$e1 \Rightarrow e2$	=-E, pr 1
2	$e2$	$\Rightarrow$-E, 3, pr 2

Case 2: $E(p)$ is an identifier different from p, say v. In this case $E(e1)$ and $E(e2)$ are both v and the theorem holds trivially.

Case 3: $E(p)$ has the form $\neg G(p)$, for some expression G. By induction, we may assume that a proof of

From $e2 = e1$, $G(e2)$ **infer** $G(e1)$

exists, and we prove the desired result as follows.

From $e1 = e2$, $\neg G(e1)$ **infer** $\neg G(e2)$

1	$e1 = e2$	pr 1
2	$\neg G(e1)$	pr 2
3	**From** $G(e2)$ **infer** $G(e1) \wedge \neg G(e1)$	
	3.1 $e2 = e1$	$\Rightarrow$-E, ex. 25 of 3.3, 1
	3.2 $(e2 = e1) \wedge G(e2) \Rightarrow G(e1)$	$\Rightarrow$-I, assumed proof
	3.3 $(e2 = e1) \wedge G(e2)$	$\wedge$-I, 3.1, pr 1
	3.4 $G(e1)$	$\Rightarrow$-E, 3.2, 3.3
	3.5 $G(e1) \wedge \neg G(e1)$	$\wedge$-I, 3.4, 2
4	$\neg G(e2)$	$\neg$-I, 3

Case 4: $E(p)$ has the form $G(p) \wedge H(p)$ for some expressions G and H. In this case, by induction we may assume that the following proofs exist.

From $e1 = e2$, $G(e1)$ **infer** $G(e2)$
From $e1 = e2$, $H(e1)$ **infer** $H(e2)$

We can then give the following proof.

From $e1 = e2$, $G(e1) \wedge H(e1)$ **infer** $G(e2) \wedge H(e2)$

1	$G(e1)$	$\wedge$-E, pr 2
2	$G(e2)$	Assumed proof, pr 1, 1
3	$H(e1)$	$\wedge$-E, pr 2
4	$H(e2)$	Assumed proof, pr 1, 3
5	$G(e2) \wedge H(e2)$	$\wedge$-I, 2, 4

The rest of the cases, where $E(p)$ has one of the forms $G(p) \vee H(p)$, $G(p) \Rightarrow H(p)$ and $G(p) = H(p)$, are left to the reader.

11. **From** $a = b$, $b = c$ **infer** $a = c$

1	$a \Rightarrow b$	=-E, pr 1
2	$b \Rightarrow c$	=-E, pr 2
3	**From** a **infer** c	
	3.1 b	$\Rightarrow$-E, 1, pr 1
	3.2 c	$\Rightarrow$-E, 2, 3.1
4	$a \Rightarrow c$	$\Rightarrow$-I, 3
5	$c \Rightarrow a$	proof omitted, similar to 1-4
6	$a = c$	=-I, 4, 5

Answers for Section 3.5

1. Conjecture 4, which can be written as $bl \wedge \neg gj \Rightarrow \neg tb$, is not valid, as can be seen by considering the state with $tb = T$, $ma = T$, $bl = T$, $gh = F$, $fd = T$ and $gj = F$.

Conjecture 8, which can be written as $gh \wedge tb \Rightarrow (bl \Rightarrow \neg fd)$, is proved as follows:

From gh, tb **infer** $bl \Rightarrow \neg fd$

1	$\neg\neg tb$	subs, Negation, pr 2
2	$\neg bl \vee ma$	(3.4.6), Premise 1, 1
3	$\neg\neg gh$	subs, Negation, pr 1
4	$\neg ma \vee \neg fd$	(3.4.6), Premise 2, 3
5	**From** bl **infer** $\neg fd$	
	5.1 $\neg\neg bl$	subs, Negation, pr 1
	5.2 ma	(3.4.6), 2, 5.1
	5.3 $\neg\neg ma$	subs, Negation, 5.2
	5.4 $\neg fd$	(3.4.6), 4, 5.3
6	$bl \Rightarrow \neg fd$	$\Rightarrow$-I, 5

2. For the proofs of the valid conjectures using the equivalence transformation system of chapter 2, we first write here the disjunctive normal form of the Premises:

Premise 1: $\neg tb \vee \neg bl \vee ma$
Premise 2: $\neg ma \vee \neg fd \vee \neg gh$
Premise 3: $gj \vee (fd \wedge \neg gh)$

Conjecture 2, which can be written in the form $(ma \wedge gh) \Rightarrow gj$, is proved as follows. First, use the laws of Implication and De Morgan to put it in disjunctive normal form:

(E.1) $\quad \neg ma \vee \neg gh \vee gj.$

To show (E.1) to be true, it is necessary to show that at least one of the disjuncts is true. Assume, then, that the first two are false: $ma = T$ and $gh = T$. In that case, Premise 2 reduces to $\neg fd$, so we conclude that $fd = F$. But then Premise 3 reduces to gj, and since Premise 3 is true, gj is true, so that (E.1) is true also.

Answers for Section 4.1.

1. (a) T. (b) T. (c) F. (d) T.

2. (a) $\{1,2,3,4,6\}$. (b) $\{2,4\}$. (c) F. (d) F.

3. (a) U. (b) T. (c) U. (d) U.

5. We don't build all the truth tables explicitly, but instead analyze the various cases.

Associativity. To prove a **cor** $(b$ **cor** $c)$ and $(a$ **cor** $b)$ **cor** c equivalent, we investigate possible values of a. Suppose $a = T$. Then evaluation using the truth table for **cor** shows that both expressions yield T. Suppose $a = F$. Then evaluation shows that both yield b **cor** c, so that they are the same. Suppose $a = U$. Then evaluation shows that both expressions yield U. The proof of the other associative law is similar.

Answers for Section 4.2

1. The empty string ε —the string containing zero characters— is the identity element, because for all strings x, $x \mid \varepsilon = x$.

4. $(N\, i: 0 \leqslant i < n: x = b[i]) = (N\, i: 0 \leqslant i < m: x = c[i])$.

5. $(A\, k: 0 \leqslant k < n:$
$\quad (N\, i: 0 \leqslant i < n: b[k] = b[i]) = (N\, j: 0 \leqslant j < n: b[k] = c[i]))$.

6. (a) $(A\, i: j \leqslant i < k+1: b[i] = 0)$

(b) $\neg(E\, i: j \leqslant i < k+1: b[i] = 0)$, or $(A\, i: j \leqslant i < k+1: b[i] \neq 0)$

(c) Some means at least one: $(E\, i: j \leqslant i < k+1: b[i] = 0)$, or, better yet, $(N\, i: j \leqslant i < k+1: b[i] = 0) > 0$

(d) $(0 \leqslant i < n$ **cand** $b[i] = 0) \Rightarrow j \leqslant i \leqslant k$, or
$(A\, i: 0 \leqslant i < n: b[i] = 0 \Rightarrow j \leqslant i \leqslant k)$

Answers for Section 4.3

1. (a) $(\overbrace{E\, k: 0 \leqslant k < n: P \wedge H_k}(T)) \wedge k > 0$ (invalid)

(b) $(\overbrace{A\, j: 0 \leqslant j < n: B_j \Rightarrow wp(SL_j, R)})$

Answers for Section 4.4

1. $E_j^i = E$ (i is not free in E)

$E_{n+1}^n = (A\, i: 0 \leqslant i < n+1: b[i] < b[i+1])$

2. E_j^i is invalid, because it yields two interpretations of j:

$E_j^i = n > j \wedge (N j: 1 \leqslant j < n: n \div j = 0) > 1$

3. $E_j^i = E$ (since i is not free in E)

4. The precondition is $x+1>0$, which can be written as R^x_{x+1} (compare with the assignment statement $x := x+1$).

6. (a) For the expressions T, F and *id* where *id* is an identifier that is not i, $E^i_e = E$.

(b) For the expression consisting of identifier i, $E^i_e = e$.

(c) $(E)^i_e = (E^i_e)$

(d) $(\neg E)^i_e = \neg (E^i_e)$

(e) $(E1 \wedge E2)^i_e = E1^i_e \wedge E2^i_e$ (Similarly for $\vee$, $=$ and $\Rightarrow$)

(f) $(\boldsymbol{A}\, i: m \leqslant i \leqslant n: E)^i_e = (\boldsymbol{A}\, i: m \leqslant i \leqslant n: E)$

For identifier j not identifier i,
$(\boldsymbol{A}\, j: m \leqslant j \leqslant n: E)^i_e = (\boldsymbol{A}\, j: m^i_e \leqslant j \leqslant n^i_e: E^i_e)$ (Similarly for $\boldsymbol{E}$ and $\boldsymbol{N}$.)

Answers for Section 4.5

1. (a) $\boldsymbol{E}$ is commutative, as is $\boldsymbol{A}$; this can be written as $(\boldsymbol{E}t: (\boldsymbol{E}p: fool(p,t)))$, or $(\boldsymbol{E}p: (\boldsymbol{E}t: fool(p,t)))$, or $(\boldsymbol{E}p,t: fool(p,t))$.

2. (b) $(\boldsymbol{A}\, a,b,c: integer(a,b,c):$
$sides(a,b,c) = a+b \geqslant c \wedge a+c \geqslant b \wedge b+c \geqslant a)$

Answers for Section 4.6

1. (a) $x=6$, $y=6$, $b=T$. (b) $x=5$, $y=5$, $b=T$.

Answers for Section 5.1

1. (a) (3, 4, 6, 8). (d) (8, 6, 4, 2).

2. (a) 0. (b) 2. (c) 0.

3. (a) $(i=j \wedge 5=5) \vee (i \neq j \wedge 5=b[j]) \;=\; (i=j) \vee b[j]=5$.
(b) $b[i]=i$.

Answers for Section 5.2

1. (Those exercises without abbreviations are not answered.)
(a) $b[j:k]=0$.
(b) $b[j:k] \neq 0$.
(c) $0 \in b[j:k]$.
(d) $0 \notin b[0:j-1] \wedge 0 \notin b[k+1:n-1]$.

2. (a) $0 \leqslant p \leqslant q+1 \leqslant n \;\wedge\; b$

0	p	q	$n-1$
$\leqslant x$			$>x$

Answers for Section 6.2

1. (a) *First specification*: $\{n>0\}\ S\ \{x=max(\{y \mid y \in b[0{:}n-1]\})\}$.

Second specification: Given fixed n and fixed array $b[0{:}n-1]$, establish

$$R\colon x=max(\{y \mid y \in b\}).$$

For program development it may be useful to replace *max* by its meaning. The result assertion R would then be

$$R\colon (\boldsymbol{E}\, i\colon 0\leqslant i<n\colon x=b[i]) \wedge (\boldsymbol{A}\, i\colon 0\leqslant i<n\colon b[i]\leqslant x)$$

(d) *First specification*:

$$\begin{array}{l}\{n>0\}\\ S\\ \{0\leqslant i<n \wedge (\boldsymbol{A}\, j\colon 0\leqslant j<n\colon\ b[i]\geqslant b[j]) \wedge b[i]>b[0{:}i-1]\}.\end{array}$$

Second specification: Given fixed $n>0$ and fixed array $b[0{:}n-1]$, set i to establish

$$R\colon 0\leqslant i<n \wedge b[0{:}n-1]\leqslant b[i] \wedge b[i]>b[0{:}i-1]$$

(k) Define $average(i)=(\Sigma j\colon 0\leqslant j<4\colon grade[i, j])$.

First specification:

$$\{n>0\}\ S\ \{0\leqslant i<n \wedge (\boldsymbol{A}\, j\colon 0\leqslant j<n\colon average(i)\geqslant average(j))\}.$$

Answers for Chapter 7

1. (a) $i+1>0$, or $i\geqslant 0$.
(b) $i+2+j-2=0$, or $i+j=0$.

3. Suppose $Q \Rightarrow R$. Then $Q \wedge R = Q$. Therefore

$$\begin{array}{lll} wp(S,Q)= & wp(S, Q\wedge R) & (\text{since } Q\wedge R=R)\\ = & wp(S,Q)\wedge wp(S,R) & (\text{by (7.4)})\\ \Rightarrow & wp(S,R) & \end{array}$$

This proves (7.5).

6. By (7.6), we see that LHS (of (7.7)) $\Rightarrow$ RHS. Hence it remains to show that RHS $\Rightarrow$ LHS. To prove this, we must show that any state s in $wp(S, Q\vee R)$ is either guaranteed to be in $wp(S, Q)$ or guaranteed to be in $wp(S, R)$. Consider a state s in $wp(S, Q\vee R)$. Because S is deterministic, execution of S beginning in s is guaranteed to terminate in a single, unique state s', with s' in $Q\vee R$. This unique state s' must be either in Q, in which case s is in $wp(S, Q)$, or in R, in which case s is in

$wp(S, R)$.

7. This exercise is intended to make the reader more aware of how quantification works in connection with wp, and the need for the rule that each identifier be used in only one way in a predicate. Suppose that $Q \Rightarrow wp(S, R)$ is true in every state. This assumption is equivalent to

(E7.1) $(\mathcal{A}\, x\colon Q \Rightarrow wp(S, R))$.

We are asked to analyze predicate (7.8): $\{(\mathcal{A}\, x\colon Q)\}\ S\ \{(\mathcal{A}\, x\colon R)\}$, which is equivalent to

(E7.2) $(\mathcal{A}\, x\colon Q) \Rightarrow wp(S, (\mathcal{A}\, x\colon R))$.

Let us analyze this first of all under the rule that no identifier be used in more than one way in a predicate. Hence, rewrite (E7.2) as

(E7.3) $(\mathcal{A}\, x\colon Q) \Rightarrow wp(S, (\mathcal{A}\, z\colon R_z^x))$.

and assume that x does not appear in S and that z is a fresh identifier. We argue operationally that (E7.3) is true. Suppose the antecedent of (E7.3) is true in some state s, and that execution of S begun in s terminates in state s'. Because S does not contain identifier x, we have $s(x) = s'(x)$.

Because the antecedent of (E7.3) is true in s, we conclude from (E7.1) that $(\mathcal{A}\, x\colon wp(S, R))$ is also true in state s. Hence, no matter what the value of x in s, $s'(R)$ is true. But $s(x) = s'(x)$. Thus, no matter what the value of x in s', $s'(R)$ is true. Hence, so is $s'((\mathcal{A}\, x\colon R))$, and so is $s'((\mathcal{A}\, z\colon R_z^x))$. Thus, the consequent of (E7.3) is true in s, and (E7.3) holds.

We now give a counterexample to show that (E7.2) need not hold if x is assigned in command S and if x appears in R. Take command S: $x := 1$. Take R: $x = 1$. Take Q: T. Then (E7.1) is

$$(\mathcal{A}\, x\colon T \Rightarrow wp(\text{“}x := 1\text{”}, x = 1))$$

which is true. But (E7.2) is false in this case: its antecedent $(\mathcal{A}\, x\colon T)$ is true but its consequent $wp(\text{“}x := 1\text{”}, (\mathcal{A}\, x\colon x = 1))$ is false because predicate $(\mathcal{A}\, x\colon x = 1)$ is F.

We conclude that if x occurs both in S and R, then (E7.2) does not in general follow from (E7.1).

Answers for Chapter 8

3. By definition, $wp(make\text{-}true, F) = T$, which violates the law of the Excluded Miracle, (7.3).

5. $wp(\text{"}S1;\ S2\text{"}, Q) \vee wp(\text{"}S1;\ S2\text{"}, R)$

$= wp(S1, wp(S2, Q)) \vee wp(S1, wp(S2, R))$ (by definition)

$= wp(S1, wp(S2, Q) \vee wp(S2, R))$ (since $S1$ satisfies (7.7))

$= wp(S1, wp(S2, Q \vee R))$ (since $S2$ satisfies (7.7))

$= wp(\text{"}S1;\ S2\text{"}, Q \vee R)$ (by definition)

Answers for Section 9.1

1. (a) $(2*y+3) = 13$, or $y = 5$.
(b) $x+y < 2*y$, or $x < y$.
(c) $0 < j+1 \wedge (\boldsymbol{A}\, i: 0 \leqslant i \leqslant j+1: b[i] = 5)$.
(d) $(b[j] = 5) = (\boldsymbol{A}\, i: 0 \leqslant i \leqslant j: b[i] = 5)$

3. Execution of $x := e$ in state s evaluates e to yield the value $s(e)$ and stores this value as the new value of x. This is our conventional model of execution. Hence, for the final state s' we have $s' = (s;\ x{:}s(e))$.

We want to show that $s'(R) = s(R^x_e)$. But this is simply lemma 4.6.2. This means that if R is to be true (false) after the assignment (i.e. in state s') then R^x_e must be true (false) before (i.e. in state s). This is exactly what definition (9.1) indicates.

4. Writing both Q and e as functions of x, we have

$$
\begin{aligned}
& wp(\text{"}x := e(x)\text{"}, sp(Q(x), \text{"}x := e(x)\text{"})) \\
& \quad = wp(\text{"}x := e(x)\text{"}, (\boldsymbol{E}\, v: Q(v) \wedge x = e(v))) \\
& \quad = (\boldsymbol{E}\, v: Q(v) \wedge x = e(v))^x_{e(x)} \\
& \quad = (\boldsymbol{E}\, v: Q(v) \wedge e(x) = e(v)) \qquad \text{(E4.1)}
\end{aligned}
$$

The last line follows because neither $Q(v)$ nor $e(v)$ contains a reference to x. Now suppose Q is true in some state s. Let $v = s(x)$, the value of x in state s. For this v, $(Q(v) \wedge e(x) = e(v))$ is true in state s, so that (E4.1) is also true in s. Hence $Q \Rightarrow (E4.1)$, which is what we needed to show.

Answers for Section 9.2

1. We prove only that $x := e1;\ y := e2$ is equivalent to $x, y := e1, e2$. Write any postcondition R as a function of x and y: $R(x,y)$.

$$
\begin{aligned}
& wp(\text{"}x := e1;\ y := e2\text{"}, R(x, y)) \\
& \quad = wp(\text{"}x := e1\text{"}, wp(\text{"}y := e2\text{"}, R(x, y))
\end{aligned}
$$

$= wp(\text{“}x := e1\text{”}, R(x, y)^{y}_{e2})$
$= wp(\text{“}x := e1\text{”}, R(x, e2))$
$= R(x, e2)^{x}_{e1}$
$= R(e1, e2)$ (since x is not free in $e2$)
$= wp(\text{“}x, y := e1, e2\text{”}, R(x, y))$ (by definition)

3. (a) $1 * c^d = c^d$, or T
(b) $1 \leqslant 1 < n \wedge b[0] = (\Sigma j: 0 \leqslant j \leqslant 0: b[j]$, or $1 < n$
(c) $0^2 < 1 \wedge (0+1)^2 \geqslant 1$, or T

4. In these, it must be remembered that x is a function of the identifiers involved. Hence, if x occurs in an expression in which a substitution is being made, that substitution may change x also. In the places where this happens, x is written as a function of the variables involved. See especially exercise (b).

(a) $wp(\text{“}a, b := a+1, x\text{”}, b = a+1) = x = a+2$. Hence, take $x = a+2$.

(b) $wp(\text{“}a := a+1;\ b := x(a)\text{”}, b = a+1)$
$= wp(\text{“}a := a+1\text{”}, x(a) = a+1)$
$= x(a+1) = a+2$

This is satisfied by taking $x(a) = a+1$. Hence, take $x = a+1$.

(e) $wp(\text{“}i := i+1;\ j := x(i)\text{”}, i = j)$
$= wp(\text{“}i := i+1\text{”}, i = x(i))$
$= i+1 = x(i+1)$

Answers for Section 9.3

1. For each part, the weakest precondition, determined by textual substitution, is given and then simplified.

(a) $(b; i{:}i)[(b; i{:}i)[i]] = i = (b; i{:}i)[i] = i$
$= i = i = T$

(b) $(\boldsymbol{E} j: i \leqslant j < n: (b; i{:}5)[i] \leqslant (b; i{:}5)[j])$
$= (\boldsymbol{E} j: i \leqslant j < n: 5 \leqslant (b; i{:}5)[j])$
$= (\boldsymbol{E} j: i < j < n: 5 \leqslant (b; i{:}5)[j]) \vee 5 \leqslant (b; i{:}5)[i]$
$= (\boldsymbol{E} j: i < j < n: 5 \leqslant (b; i{:}5)[j]) \vee 5 \leqslant 5$
$= T$

Answers for Section 9.4

1. (a) $R^{b, x}_{(b; i:e;\ j:f),\, g}$

2. For each part, the weakest precondition determined by textual substitution is given and then simplified.

(a) $(b;\ i{:}3;\ 2{:}4)[i]=3$
$= (i=2 \wedge 4=3) \vee (i \neq 2 \wedge 3=3)$
$= i \neq 2$

(g) $b[p]=(b;\ p{:}b[b[p]];\ b[p]{:}p)[(b;\ p{:}b[b[p]];\ b[p]{:}p)[b[p]]]$
$= b[p]=(b;\ p{:}b[b[p]];\ b[p]{:}p)[p]$
$= (p=b[p] \wedge b[p]=p) \vee (p \neq b[p] \wedge b[p]=b[b[p]])$
$= p=b[p] \vee b[p]=b[b[p]]$ (see (c))

5. The lemma has been proven for the case that $\bar{x}$ consists of distinct identifiers, and we need only consider the case that $\bar{x} = b \circ s_1, \ldots, b \circ s_n$. To prove this case, we will need to use the obvious fact that

(E5.1) $(b;\ s{:}b \circ s)=b$

Remembering that x_i *equiv* $b{:}s_i$, we have

$$\begin{aligned}(E^{\bar{x}}_{\bar{u}})^{\bar{u}}_{\bar{x}} &= (E^{b}_{(b;\ s_1:u_1;\ \cdots;\ s_n:u_n)})^{\bar{u}}_{\bar{x}} \\ &= E^{b}_{(b;\ s_1:b\circ s_1;\ \cdots;\ s_n:b\circ s_n)} \quad \text{(substitute } x_i \text{ for each } u_i\text{)} \\ &= E^{b}_{b} \quad \text{(n applications of (E5.1))} \\ &= E\end{aligned}$$

Answers for Chapter 10

3. Letting $R \ = \ q^*w+r=x \wedge r \geqslant 0$, we have

$$\begin{aligned}wp(S3,\ R) &= (w \leqslant r \vee w > r) \wedge \\ &\qquad (w \leqslant r \Rightarrow wp(\text{“}r,\ q := r-w,\ q+1\text{”},\ R)) \wedge \\ &\qquad (w > r \Rightarrow wp(skip,\ R)) \\ &= (w \leqslant r \Rightarrow ((q+1)^*w+r-w=x \wedge r-w \geqslant 0)) \wedge (w > r \Rightarrow R) \\ &= (w \leqslant r \Rightarrow q^*w+r=x \wedge r-w \geqslant 0) \wedge (w > r \Rightarrow R)\end{aligned}$$

This is implied by R.

6.
$$\begin{aligned}wp(S6,\ R) &= (f[i]<g[j] \vee f[i]=g[j] \vee f[i]>g[j]) \wedge \\ &\qquad (f[i]<g[j] \Rightarrow R^{i}_{i+1}) \wedge \\ &\qquad (f[i]=g[j] \Rightarrow R) \wedge \\ &\qquad (f[i]>g[j] \Rightarrow R^{j}_{j+1}) \\ &= R \wedge (f[i]<g[j] \Rightarrow f[i+1] \leqslant X) \wedge (f[i]>g[j] \Rightarrow g[j] \geqslant X) \\ &= R \quad \text{(since } R \text{ implies that } g[j] \leqslant X \text{ and } f[i] \leqslant X\text{)}\end{aligned}$$

Answers for Chapter 11

2. In the proof of theorem 10.5 it was proven that

$$(\boldsymbol{A}\, i:\ P \wedge B_i \Rightarrow wp(S_i, P)) \ = \ P \wedge (\boldsymbol{A}\, i:\ B_i \Rightarrow wp(S_i, P)).$$

Therefore, given assumption 1, we have

$$\begin{aligned} P \wedge \mathrm{BB} &= P \wedge \mathrm{BB} \wedge T \\ &= P \wedge \mathrm{BB} \wedge (\boldsymbol{A}\, i\colon P \wedge B_i \Rightarrow wp(S_i, P)) \text{ (since 1. is true)} \\ &= P \wedge \mathrm{BB} \wedge P \wedge (\boldsymbol{A}\, i\colon B_i \Rightarrow wp(S_i, P)) \\ &\Rightarrow \mathrm{BB} \wedge (\boldsymbol{A}\, i\colon B_i \Rightarrow wp(S_i, P)) \\ &= wp(\mathrm{IF}, P) \end{aligned}$$

3. By a technique similar to that used in exercise 2, we can show that assumption 3 of theorem 11.6 implies

(E3.1) $\quad P \wedge \mathrm{BB} \Rightarrow wp(\text{“}T := t;\ \mathrm{IF}\text{”}, t < T)$

Thus, we need only show that (E3.1) implies 3' of theorem 11.6. Note that P, IF, and t do not contain $t1$ or $t0$. Since IF does not refer to T and $t0$, we know that $wp(\mathrm{IF}, t1 \leqslant t0+1) = \mathrm{BB} \wedge t1 \leqslant t0+1$. We then have the following:

$$\begin{aligned} (E3.1) &= P \wedge \mathrm{BB} \Rightarrow wp(\mathrm{IF}, t < t1)_t^{t1} \\ &\qquad \text{(by definition of :=)} \\ &\Rightarrow P \wedge \mathrm{BB} \wedge t \leqslant t0+1 \Rightarrow wp(\mathrm{IF}, t \leqslant t1-1)_t^{t1} \wedge t \leqslant t0+1 \\ &\qquad \text{(Insert } t \leqslant t0+1 \text{ on both sides of } \Rightarrow\text{)} \\ &= P \wedge \mathrm{BB} \wedge t \leqslant t0+1 \Rightarrow wp(\mathrm{IF}, t \leqslant t1-1)_t^{t1} \wedge (t1 \leqslant t0+1)_t^{t1} \\ &= P \wedge \mathrm{BB} \wedge t \leqslant t0+1 \Rightarrow (wp(\mathrm{IF}, t \leqslant t1-1) \wedge t1 \leqslant t0+1)_t^{t1} \\ &\qquad \text{(Distributivity of textual substitution)} \\ &= P \wedge \mathrm{BB} \wedge t \leqslant t0+1 \Rightarrow (wp(\mathrm{IF}, t \leqslant t1-1) \wedge wp(\mathrm{IF}, t1 \leqslant t0+1))_t^{t1} \\ &\qquad \text{(IF does not contain } t1 \text{ nor } t0\text{)} \\ &= P \wedge \mathrm{BB} \wedge t \leqslant t0+1 \Rightarrow wp(\mathrm{IF}, t \leqslant t1-1 \wedge t1 \leqslant t0+1)_t^{t1} \\ &\qquad \text{(Distributivity of Conjunction)} \\ &= P \wedge \mathrm{BB} \wedge t \leqslant t0+1 \Rightarrow wp(\mathrm{IF}, t \leqslant t0)_t^{t1} \\ &= P \wedge \mathrm{BB} \wedge t \leqslant t0+1 \Rightarrow wp(\text{“}t1 := t;\ \mathrm{IF}\text{”}, t \leqslant t0) \\ &= P \wedge \mathrm{BB} \wedge t \leqslant t0+1 \Rightarrow wp(\text{“IF”}, t \leqslant t0) \end{aligned}$$

Since the derivation holds irrespective of the value $t0$, it holds for all $t0$, and 3' is true.

4. We first show that (11.7) holds for $k = 0$ by showing that it is equivalent to assumption 2:

$$\begin{aligned} &P \wedge \mathrm{BB} \Rightarrow t > 0 && \text{(Assumption 2)} \\ &= \neg P \vee \neg \mathrm{BB} \vee t > 0 && \text{(Implication, De Morgan)} \\ &= \neg P \vee \neg(t \leqslant 0) \vee \neg \mathrm{BB} \\ &= P \wedge t \leqslant 0 \Rightarrow \neg \mathrm{BB} && \text{(De Morgan, Implication)} \\ &= P \wedge t \leqslant 0 \Rightarrow P \wedge \neg \mathrm{BB} \end{aligned}$$

$$= P \wedge t \leqslant 0 \Rightarrow H_0(P \wedge \neg \mathrm{BB}) \quad \text{(Definition of } H_0\text{)}$$

Assume (11.7) true for $k = K$ and prove it true for $k = K+1$. We have:

$$\begin{aligned} P \wedge \mathrm{BB} \wedge t \leqslant K+1 &\Rightarrow wp(\mathrm{IF}, P \wedge t \leqslant K) && \text{(this is 3')} \\ &\Rightarrow wp(\mathrm{IF}, H_K(P \wedge \neg \mathrm{BB})) && \text{(Induction hyp.)} \end{aligned}$$

and
$$\begin{aligned} P \wedge \neg \mathrm{BB} \wedge t \leqslant K+1 &\Rightarrow P \wedge \neg \mathrm{BB} \\ &= H_0(P \wedge \neg \mathrm{BB}) \end{aligned}$$

These two facts yield

$$\begin{aligned} P \wedge t \leqslant K+1 &\Rightarrow H_0(P \wedge \neg \mathrm{BB}) \vee wp(\mathrm{IF}, P \wedge \neg \mathrm{BB}) \\ &= H_{K+1}(P \wedge \neg \mathrm{BB}) \end{aligned}$$

which shows that (11.7) holds for $k = K+1$. By induction, (11.7) holds for all k.

6. $H'_0(R) = \neg \mathrm{BB} \wedge R$. For $k > 0$, $H'_k(R) = wp(\mathrm{IF}, H'_{k-1}(R))$. $(\boldsymbol{E}k\colon 0 \leqslant k\colon H'_k(R))$ represents the set of states in which DO will terminate with R true in *exactly* k iterations. On the other hand, $wp(\mathrm{DO}, R)$ represents the set of states in which DO will terminate with R true in k *or less* iterations.

10. (1) $wp(\text{“}i := 1\text{”}, P) = 0 < 1 \leqslant n \wedge (\boldsymbol{E}p\colon 1 = 2^p)$
$= T$ (above, take $p = 0$).

(2) $wp(S_1, P) = wp(\text{“}i := 2*i\text{”}, 0 < i \leqslant n \wedge (\boldsymbol{E}p\colon i = 2^p))$
$= 0 < 2*i \leqslant n \wedge (\boldsymbol{E}p\colon 2*i = 2^p)$,

which is implied by $P \wedge 2*i \leqslant n$.

(3) $P \wedge \neg \mathrm{BB} = 0 < i \leqslant n \wedge (\boldsymbol{E}p\colon i = 2^p) \wedge 2*i > n$,
which is equivalent to R.

(4) $P \wedge \mathrm{BB} \Rightarrow 0 < i \leqslant n \wedge 2*i \leqslant n$
$\Rightarrow n - i > 0$, which is $t > 0$.

(5) $wp(\text{“}t1 := t;\ S_1\text{”}, t < t1)$
$= wp(\text{“}t1 := n - i;\ i := 2*i\text{”}, n - i < t1)$
$= wp(\text{“}t1 := n - i\text{”}, n - 2*i < t1)$
$= n - 2*i < n - i$
$= -i < 0$, which is implied by P.

Answers for Chapter 12

1. We have the following equivalence transformations:

$$\begin{aligned} \{T(u)\}\ S\ \{R\} &= (\boldsymbol{A}u\colon \{T(u)\}\ S\ \{R\}) \\ &= (\boldsymbol{A}u\colon T(u) \Rightarrow wp(S, R)) \\ &= (\boldsymbol{A}u\colon \neg T(u) \vee wp(S, R)) \quad \text{(Implication)} \end{aligned}$$

$$
\begin{aligned}
&= (\boldsymbol{A}\, u\colon \neg T(u)) \vee wp(S, R) \\
&\quad \text{(Since neither } S \text{ nor } R \text{ contains } u\text{)} \\
&= \neg(\boldsymbol{E}\, u\colon T(u)) \vee wp(S, R) \\
&= (\boldsymbol{E}\, u\colon T(u)) \Rightarrow wp(S, R) \qquad \text{(Implication)} \\
&= \{(\boldsymbol{E}\, u\colon T(u))\}\ S\ \{\} \qquad \text{(Implication)}
\end{aligned} \tag{E1.1}
$$

The quantifier $\boldsymbol{A}$ in the precondition of (12.7) of theorem 12.6 is necessary; without it, predicate (12.7) has a different meaning. With the quantifier, the predicate can be interpreted as follows: The procedure call can be executed in a state s to produce the desired result R if all *possible* assignments $\bar{u}$, $\bar{v}$ to the result parameters and arguments establish the truth of R. Without the quantifier, the above equivalence indicates that an existential quantifier is implicitly present. With this implicit existential quantifier, the predicate can be interpreted as follows: The procedure call can be executed in a state s to produce the desired result R if there exists at least one possible assignment of values $\bar{u}$, $\bar{v}$ that establishes the truth of R. But, since there is no quarantee that this one possible set of values $\bar{u}$, $\bar{v}$ will actually be assigned to the parameters and arguments, this statement is generally false.

Answers for Chapter 14

1. (b) Q: $x = X$. $\quad R$: $(X \geqslant 0 \wedge x = X) \vee (X \leqslant 0 \wedge x = -X)$.

if $x \geqslant 0 \rightarrow$ *skip* [] $x \leqslant 0 \rightarrow x := -x$ **fi**.

2. Assume the next highest permutation exists (it doesn't for example, for d = 543221). In the following discussion, it may help to keep the example

$$
\begin{aligned}
d &= (1, 2, 3, 5, 4, 2) \\
d' &= (1, 2, 4, 3, 2, 5)
\end{aligned}
$$

in mind. There is a least integer i, $0 \leqslant i < n$, such that $d[0{:}i-1] = d'[0{:}i-1]$ and $d[i] < d'[i]$. One can show that i is well-defined by the fact that $d[i+1{:}n-1]$ is a non-increasing sequence and that $d[i] < d[i+1]$.

In order for d' to be the next highest permutation, $d'[i]$ must contain the smallest value of $d[i+1{:}n-1]$ that is greater than $d[i]$. Let the rightmost element of $d[i+1{:}n-1]$ with this value be $d[j]$. Consider $d'' = (d;\ i{:}d[j];\ j{:}d[i])$. d'' represents the array d but with the values at positions i and j interchanged. In the example above, $d'' = (1, 2, 4, 5, 3, 2)$. Obviously, d'' is a higher permutation than d, but perhaps not the *next* highest. Moreover, $d'[0{:}i] = d''[0{:}i]$.

It can be proved that $d''[i+1{:}n-1]$ is a non-increasing sequence. Hence, reversing $d''[i+1{:}n-1]$ makes it an increasing sequence and, therefore, as small as possible. This yields the desired next highest permutation d.

To summarize, let i and j satisfy, respectively,

$$0 \leqslant i < n-1 \wedge d[i] < d[i+1] \wedge d[i+1:n-1] \text{ is non-increasing}$$
$$i < j < n \wedge d[j] > d[i] \wedge d[j+1:n-1] \leqslant d[i]$$

Introducing the notation $reverse(b, f, g)$ to denote the array b but with $b[f:g]$ reversed, we then have that the next highest permutation d' is

$$d' = reverse((d; i:b[j]; j:b[i]),\ i+1,\ n-1).$$

The algorithm is then: calculate i; calculate j; swap $b[i]$ and $b[j]$; reverse $b[i+1:n-1]$! Here, formalizing the idea of a next highest permutation leads directly to an algorithm to calculate it!

Answers for Section 15.1

3. $i, x := 1, b[0]$;
do $i \neq n \rightarrow$ **if** $x \geqslant b[i] \rightarrow i, x := i+1, b[i]$
[] $x \leqslant b[i] \rightarrow i := i+1$
fi
od

Answers for Section 15.2

2. The initialization is $x, y := X, Y$. Based on the properties given, the obvious commands to try are $x := x+y$, $x := x-y$, $x := y-x$, etc. Since $Y > 0$, the first one never reduces the bound function t: $x+y$, so it need not be used. The second one reduces it, but maintains the invariant only if $x > y$. Thus we have the guarded command $x > y \rightarrow x := x-y$. Symmetry encourages also the use of $y > x \rightarrow y := y-x$ and the final program is

$x, y := X, Y$;
do $x > y \rightarrow x := x-y$
[] $y > x \rightarrow y := y-x$
od
$\{0 < x = y \wedge gcd(x, y) = gcd(X, Y)\}$
$\{x = gcd(X, Y)\}$

5. $t := 0$;
do $j \neq 80$ **cand** $b[j] \neq ' ' \rightarrow t, s[t+1], j := t+1, b[j], j+1$
[] $j = 80 \rightarrow read(b); j := 0$
od

Answers for Section 16.2

2. Delete the conjunct $n < 2*i$ from R:

$\{Q: 0 < n\}$
$i := 1;$
$\{inv: 0 < i \leq n \wedge (\boldsymbol{E}p: 2^p = i)\}$
$\{bound: n - i$ (actually, $\log(n-i)$ will do)$\}$
do $2*i \leq n \rightarrow i := 2*i$ **od**
$\{R: 0 \leq i \leq n < 2*i \wedge (\boldsymbol{E}p: 2^p = i)\}$

4. Delete the conjunct $x = b[i,j]$ from R:

$i, j := 0, 0;$
$\{inv: 0 \leq i < m \wedge 0 \leq j < n \wedge x \notin b[0{:}i-1, 0{:}n-1] \wedge$
$\quad x \notin b[i, 0{:}j-1] \wedge x \in b\}$
$\{bound: (m-i)*n-j\}$
do $x \neq b[i,j] \wedge j \neq n-1 \rightarrow j := j+1$
$[]\ x \neq b[i,j] \wedge j = n-1 \rightarrow i, j := i+1, 0$
od

Answers for Section 16.3

4. (a)
$i, j := 1, n;$
$\{inv: 1 \leq i < j \leq n \wedge b[i] \leq x < b[j]\}$
$\{bound: \log(j-i)\}$
do $i+1 \neq j \rightarrow e := (i+j) \div 2;$
if $b[e] \leq x \rightarrow i := e$ $[]$ $b[e] > x \rightarrow j := e$ **fi**
od

The obvious choice for the second part of the problem is to embed the program for the first part in an alternative command:

if $x < b[1]$ $\rightarrow i := 0$
$[]\ b[1] \leq x < b[n]$ $\rightarrow$ The program (a)
$[]\ b[n] \leq x$ $\rightarrow i := n$
fi

However, there is a simpler way. Assume the existence of $b[0]$, which contains the value $-\infty$, and $b[n+1]$, which contains the value $+\infty$. As long as the program never references these values, this assumption may be made. Then, with a slight change in initialization, the program for the first part used —it even works when the array is empty, setting j to 1 in that case.

```
i, j := 0, n+1;
{inv: 0 ≤ i < j ≤ n+1 ∧ b[i] ≤ x < b[j]}
{bound: log(j−i)}
do i+1 ≠ j → e := (i+j) ÷ 2;
             {1 ≤ e ≤ n}
             if b[e] ≤ x → i := e [] b[e] > x → j := e fi
od
```

10.
```
i, p := 0, 0;
{inv: see exercise 10; bound: n−i}
do i ≠ n → Increase i, keeping invariant true:
           j := i+1;
           {inv: b[i:j−1] are all equal; bound: n−j}
           do j ≠ n cand b[j] = b[i] → j := j+1 od;
           p := max(p, j−i);
           i := j
od
```

Answers for Section 16.5

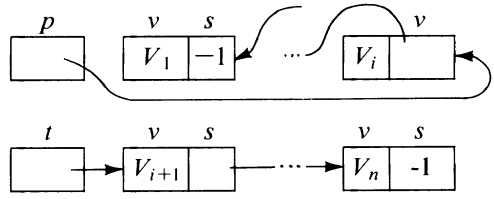
4. The only differences in this problem and exercise 3 are that the value used to separate the array into two sections is already in the array and that that value must be placed in $b[p]$. The invariant of the loop (except for the fact that $b[1] = B[1]$ and that b is a permutation of B) given in the procedure below is:

P: $m < q \leq p+1 \leq n \wedge x = B[1] \wedge b$

	m	$m+1$	q	p	$n-1$
	x	$\leq x$	?		$> x$

```
proc Partition(value result b: array [*] of integer;
               value m, n: integer; result k: integer);
  var x, q, p: integer;
  begin
    x, q, p := b[m], m+1, n−1;
    {inv: P; bound: p−q+1}
    do q ≤ p → if b[q] ≤ x          → q := q+1
               [] b[p] > x          → p := p−1
               [] b[q] > x ≥ b[p]   → b[q], b[p] := b[p], b[q];
                                      q, p := q+1, p−1
               fi
    od; {p = q−1 ∧ b[m+1:p] ≤ b[m] ∧
         b[p+1:n−1] > b[m] ∧ b[m] = B[m] = x}
    b[m], b[p] := b[p], b[m]
  end
```

6. The precondition states that the linked list is in order; the postcondition that it is reversed. This suggests an algorithm that at each step reverses one link: part of the list is reversed and part of it is in order. Thus, using another variable t to point to the part of the list that is in order, the invariant is



Initially, the reversed part of the list is empty and the unreversed part is the whole list. This leads to the algorithm

$p, t := -1, p;$
do $t \neq -1 \rightarrow p, t, s[t] := t, s[t], p$ **od**

8. The precondition and postconditions are

Q: $x \in b[0{:}m-1, 0{:}n-1]$
R: $0 \leqslant i < m \wedge 0 \leqslant j < n \wedge x = b[i, j]$

Actually, Q and R are quite similar, in that both state that x is in a rectangular section of b —in R, the rectangular section just happens to have only one row and column. So perhaps an invariant can be used that indicates that x is in a rectangular section of b:

$$P\text{: } 0 \leqslant i \leqslant p < m \wedge 0 \leqslant q \leqslant j < n \wedge x \in b[i{:}p, q{:}j]$$

To make progress towards termination the rectangle must be made smaller, and there are four simple ways to do this: $i := i+1$, etc. Thus, we try a loop of the form

$i, p, q, j := 0, m-1, 0, n-1;$
do $? \rightarrow i := i+1$
 [] $? \rightarrow p := p-1$
 [] $? \rightarrow q := q+1$
 [] $? \rightarrow j := j-1$
od

What could serve as guards? Consider $i := i+1$. Its execution will maintain the invariant if x is not in row i of b. Since the row is ordered, this can be tested with $b[i, j] < x$, for if $b[i, j] > x$, so are all values in row i. In a similar fashion, we determine the other guards:

```
i, p, q, j := 0, m−1, 0, n−1;
do b[i,j] <x   → i := i+1
 [] b[p, q]>x  → p:= p−1
 [] b[p, q]<x  → q:= q+1
 [] b[i, j] >x → j := j−1
od
```

In order to prove that the result is true upon termination, only the first and last guards are needed. So the middle guarded commands can be deleted to yield the program

```
i, j := 0, n−1;
do b[i,j]<x  → i := i+1
 [] b[i,j]>x → j := j−1
od {x = b[i,j]}
```

This program requires at most $n+m$ comparisons. One cannot do much better than this, for the following reason. Assume the array is square: $m = n$. The off-diagonal elements $b[0{:}m-1]$, $b[1, m-2]$, ..., $b[m-1{:}0]$ form an unordered list. Given the additional information that x is on the off-diagonal, in the worst case a minimum of m comparisons is necessary.

Answers for Section 18.2

1. Algorithm (18.2.5) was developed under the general hope that each iteration of the loop would decrease the total number of elements S (say) in the partitions still to be sorted, and so S is a first approximation to the bound function. However, a partition described in set s may be empty, and choosing an empty partition and deleting it from s does not decrease S.

Consider the pair $(S, |s|)$, where $|s|$ is the number of elements in s. Execution of the body with the first alternative of the alternative command being executed decreases the tuple (lexicographically speaking) because it decreases $|s|$. Execution with the second alternative being executed also decreases it —even though $|s|$ is increased by 1, S is decreased by at least one. Hence, a bound function is $2*S + |s|$.

Answers for Section 18.3

3. Define

$$postorder(p) = \begin{cases} empty(p) \rightarrow () \\ \neg\, empty \rightarrow (postorder(left[p]) \mid \\ \qquad\qquad postorder(right[p]) \mid (root(p)) \end{cases}$$

An iterative formulation of postorder traversal is slightly more complicated than the corresponding ones of preorder and inorder traversal. It must be remembered whether the right subtree of a tree in s has been visited. Since (pointers to) nodes of trees are represented by nonnegative integers, we make the distinction using the sign bit.

The postcondition of the program is

(E3.1) R: $c = \#p \wedge postorder(p) = b[0{:}c-1]$

Before stating the invariant, we indicate the postorder traversal of a signed integer q that represents a tree:

$$post(q) = \begin{cases} q < -1 \rightarrow root(abs(q)-2) \\ q = -1 \rightarrow () \\ q \geqslant 0 \rightarrow postorder(right[q]) \mid root(q) \end{cases}$$

Using a sequence variable s, the invariant is:

(E3.2) P: $0 \leqslant c \wedge q \geqslant -1 \wedge$
$postorder(p) = b[0{:}c-1] \mid postorder(q) \mid$
$post(s[0]) \mid \cdots \mid post(s[|s|-1])$

The program is then

$c, q, s := 0, p, ();$
$\{$*invariant*: (E3.2)$\}$
do $q \neq -1 \rightarrow q, s := left[q], (q) \mid s$
$[]\ q = -1 \wedge s \neq () \rightarrow q, s := s[0], s[1..]$
if $q < -1 \rightarrow q, c, b[c] := 0, c+1, root[-q-2]$
$[]\ q = -1 \rightarrow skip$
$[]\ q > -1 \rightarrow q, s := right[q], (-q-2) \mid s$
fi
od $\{R\}$

Answers for Section 19.2

1. *P1* is easily established using $i, x := 0, 0$. If $(xv[i], yv[i])$ is a solution to (19.2.1), then

$$r = xv[i]^2 + yv[i]^2 \leqslant 2 * xv[i]^2$$

Hence, all solutions $(xv[i], yv[i])$ to the problem satisfy $r \leqslant 2 * xv[i]^2$. Using the Linear Search Principle, we write an initial loop to determine the smallest x satisfying $r \leqslant 2 * x^2$, and the first approximation to the program is

```
i, x := 0, 0; do r > 2*x^2 → x := x+1 od;
{inv: P1 ∧ r ≤ 2*x^2}
do x^2 ≤ r → Increase x, keeping invariant true od
```

In order to increase x and keep *P1* true, it is necessary to determine if a suitable y exists for x and to insert the pair (x, y) in the arrays if it does. To do this requires first finding the value y satisfying

(E1.1) $x^2+y^2 \leqslant r \ \wedge \ x^2+(y+1)^2 > r$

Taking the second conjunct,

P2: $x^2+(y+1)^2 > r$

as the invariant of an inner loop, rewrite the program as

```
i, x := 0, 0; do r > 2*x^2 → x := x+1 od;
{inv: P1 ∧ r ≤ 2*x^2}
do x^2 ≤ r →
    Increase x, keeping invariant true:
        Determine y to satisfy (E1.1):
            y := x;
            {inv: P2}
            do x^2+y^2 > r → y := y−1 od;
        if x^2+y^2 = r → xv[v], yv[v], i, x := x, y, i+1, x+1
        [] x^2+y^2 < r → x := x+1
        fi
od
```

Now note that execution of the body of the main loop does not destroy *P2*, and therefore *P2* can be taken out of the loop. Rearrangement then leads to the more efficient program

```
i, x := 0, 0;
do r > 2*x^2 → x := x+1 od;
y := x;
{inv: P1 ∧ P2}
do x^2 ≤ r →
    Increase x, keeping invariant true:
        Determine y to satisfy (E1.1):
            do x^2+y^2 > r → y := y−1 od;
        if x^2+y^2 = r → xv[v], yv[v], i, x := x, y, i+1, x+1
        [] x^2+y^2 < r → x := x+1
        fi
od
```

Answers for Section 19.3

2. Program 19.3.2, which determines an approximation to the square root of a nonnegative integer n, is

$\{n \geqslant 0\}$
$a, c := 0, 1;$ **do** $c^2 \leqslant n \rightarrow c := 2*c$ **od;**
$\{inv: a^2 \leqslant n < (a+c)^2 \wedge (E p: 1 \leqslant p: c = 2^p)\}$
$\{bound: \sqrt{n} - a\}$
do $c \neq 1 \rightarrow c := c/2;$
if $(a+c)^2 \leqslant n \rightarrow a := a+c$
[] $(a+c)^2 > n \rightarrow skip$
fi
od
$\{a^2 \leqslant n < (a+1)^2\}$

We attempt to illustrate how a change of representation to eliminate squaring operations could be discovered.

Variables a and c are to be represented by other variables and eliminated, so that no squaring is necessary. As a first step, note that the squaring operation c^2 must be performed in some other fashion. Perhaps a fresh variable p can be used, which will always satisfy the relation

$$p = c^2$$

Now, which operations involving c can be replaced easily by operations involving p instead?

Command $c := 1$ can be replaced by $p := 1$, expression c^2 by p, $c := 2*c$ by $p := 4*p$, $c \neq 1$ by $p \neq 1$ and $c := c/2$ by $p := p/4$.

The remaining operations to rewrite are $a := 0$ (if necessary), $a := a+c$, $(a+c)^2 \leqslant n$ and $(a+c)^2 > n$. Consider the latter two expressions, which involve squaring. Performing the expansion $(a+c)^2 = (a^2 + 2*a*c + c^2)$ isolates another instance of c^2 to be replaced by p, so we rewrite the first of these as

(E3.1) $\quad a^2 + 2*a*c + p - n \leqslant 0$

Expression (E3.1) must be rewritten, using new variables, in such a way that the command $a := a+c$ can also be rewritten. What are possible new variables and their meaning?

There are a number of possibilities, for example $q = a^2$, $q = a*c$, $q = a^2 - n$, and so forth. The definition

(E3.2) $q = a*c$

is promising, because it lets us replace almost all the operations involving a. Thus, before the main loop, q will be 0 since a is 0 there. Secondly, to maintain (E3.2) across $c := c / 2$ we can insert $q := q / 2$. Thirdly, (E3.2) maintained across execution of the command $a := a + c$ by assigning a new value to q —what is the value?

To determine the value x to assign to q, calculate

$$wp(\text{“}a,\ q := a + c,\ x\text{”},\ q = a*c) \ = \ x = (a + c)*c$$

The desired assignment is therefore

$q := (a + c)*c$, which is equivalent to
$q := a*c + c^2$, which is equivalent to
$q := q + p$

With this representation, (E3.1) becomes

(E3.3) $a^2 + 2*q + p - n \leqslant 0$

Now try a third variable r to contain the value $n - a^2$, which will always be $\geqslant 0$. (E3.3) becomes

$$2*q + p - r \leqslant 0$$

And, indeed, the definition of r can also be maintained easily.

To summarize, use three variables p, q and r, which satisfy

$$p = c^2, \quad q = a*c, \quad r = n - a^2$$

and rewrite the program as

$\{n \geqslant 0\}$
$p, q, r := 1, 0, n$; **do** $p \leqslant n \rightarrow p := 4*p$ **od**;
do $p \neq 1 \rightarrow p := p/4;\ q := q/2;$
if $2*q + p \leqslant r \rightarrow q, r := q + p, ;\ r - 2*q - p$
[] $2*q + p > r \rightarrow skip$
fi
od
$\{q^2 \leqslant n < (q+1)^2\}$

Upon termination we have $p = 1$, $c = 1$, and $q = a*c = a$, so that the desired result is in q. Not only have we eliminated squaring, but all multiplications and divisions are by 2 and 4; hence, they could be implemented with shifting on a binary machine. Thus, the approximation to the square root can be performed using only adding, subtracting and shifting.

Answers for Chapter 20

1. Call a sequence (of zeroes and ones) that begins with 0000 (4 zeroes) and satisfies property 2 a *good* sequence. Call a sequence with k bits a k-sequence.

Define an ordering among good sequences as follows. Sequence *s1* is less than sequence *s2*, written $s1 .<. s2$, if, when viewed as decimal numbers with the decimal point to the extreme left, *s1* is less than *s2*. For example, $101 .<. 1011$ because $.101 < .1011$. In a similar manner, we write $101 .=. 101000$, because $.101 = .101000$. Appending a zero to a sequence yields an equal sequence; appending a one yields a larger sequence.

Any *good* sequence s to be printed satisfies $0 .<. s .<. 00001$, and must begin with 00000.

The program below iteratively generates, in order, all *good* sequences satisfying $0 .\leqslant. s .\leqslant. 00001$, printing the 36-bit ones as they are generated. The sequence currently under consideration will be called s. There will be no variable s; it is just a name for the sequence currently under consideration. s always contains at least 5 bits. Further, to eliminate problems with equal sequences, we will always be sure that s is the longest *good* sequence equal to itself.

PI: $good(s) \wedge \neg good(s \mid 0) \wedge 5 \leqslant |s| \wedge 0 .\leqslant. s .\leqslant. 00001 \wedge$
All *good* sequences $.<. s$ are printed

Sequence s with n bits could be represented by a bit array. However, it is better to represent s by an integer array $c[4{:}n-1]$, where $c[i]$ is the decimal representation of the 5-bit subsequence of s ending in bit i. Thus, we will maintain as part of the invariant of the main loop the assertion

$$P2\colon\ 5 \leqslant n = |\,s\,| \leqslant 36 \wedge$$
$$c[i] = s[i-4]*2^4 + s[i-3]*2^3 + s[i-2]*2^2 + s[i-1]*2 + s[i]$$
$$(\text{for } 4 \leqslant i < n)$$

Further, in order to keep track of which 5-bit subsequences s contains, we use a Boolean array $in[0{:}31]$:

$$P3\colon\ (\boldsymbol{A}i\colon\ 0 \leqslant i < 32\colon\ in[i] = (i \in c[4{:}n-1]))$$

With this introduction, the program should be easy to follow.

```
n, c[4],in[0]:= 5, 0, T;
in[1:31]:= F;     {s = (0,0,0,0,0)}
{inv: P1 ∧ P2 ∧ P3 ∧ ¬good(s | 0)}
do c[4] ≠ 1 →
  if n = 36 → Print sequence s
  [] n ≠ 36 → skip
  fi;
  Change s to next higher good sequence:
    do in[(c[n-1]*2 + 1) mod 32]  {(i.e. ¬good(s | 1)}
        → Delete ending 1's from s:
            do odd(c[n-1]) → n:= n-1; in[c[n]]:= F od;
          Delete ending 0:
            n:= n-1; in[c[n]]:= F
    od;
    Append 1 to s:
      c[n]:= (c[n-1]*2 + 1) mod 32; in[c[n]]:= T; n:= n+1
od
```

7. The result assertion is

$$R\colon\ c = (\boldsymbol{N}i\colon\ 0 \leqslant i < F\colon\ f[i] \notin g[0{:}G-1]) +$$
$$(\boldsymbol{N}j\colon\ 0 \leqslant j < G\colon\ g[j] \notin f[0{:}F-1])$$

We would expect to write a program that sequences up the two arrays together, in some synchronized fashion, performing a count as it goes. Thus, it makes sense to develop an invariant by replacing the two constants F and G of R as follows:

$$0 \leqslant h \leqslant F \wedge 0 \leqslant k \leqslant G \wedge$$
$$c = (\boldsymbol{N}i\colon\ 0 \leqslant i < h\colon\ f[i] \notin g[0{:}G-1]) +$$
$$(\boldsymbol{N}j\colon\ 0 \leqslant j < k\colon\ g[j] \notin f[0{:}F-1])$$

Now, consider execution of $h:= h+1$. Under what conditions does its execution leave P true? The guard for this command must obviously imply $f[h] \notin g[0{:}G-1]$, but we want the guard to be simple. As it

stands, this seems out of the question.

Perhaps strengthening the invariant will allow us to find a simple job. One thing we haven't tried to exploit is moving through the arrays in a synchronized fashion —the invariant does not imply this at all. Suppose we add to the invariant the conditions $f[h-1]<g[k]$ and $g[k-1]<g[h]$ —this might provide the synchronized search that we desire. That is, we use the invariant

$$\begin{aligned} P\colon\ & 0\leqslant h\leqslant F \wedge 0\leqslant k\leqslant G \wedge f[h-1]<g[k] \wedge g[k-1]<f[h] \\ & c = (\mathbf{N}i\colon 0\leqslant i<h\colon f[i]\notin g[0{:}G-1]) + \\ & \qquad (\mathbf{N}j\colon 0\leqslant j<k\colon g[j]\notin f[0{:}F-1]) \end{aligned}$$

Then the additional condition $f[h]<g[k]$ yields

$$g[k-1]<f[h]<g[k]$$

so that $f[h]$ does not appear in G, and increasing h will maintain the invariant. Similarly the guard for $k := k+1$ will be $g[k]<f[h]$.

This gives us our program, written below. We assume the existence of virtual values $f[-1]=g[i-1]=-\infty$ and $f[F]=g[G]=+\infty$; this allows us to dispense with worries about boundary conditions in the invariant.

```
h, k, c := 0, 0, 0;
{inv: P; bound: F−p + G−q}
do f ≠ F ∧ g ≠ G →
     if f[h]<g[k] → h, c := h+1, c+1
     [] f[h]=g[k] → h, k := h+1, k+1
     [] f[h]>g[k] → k, c := k+1, c+1
     fi
od;
Add to c the number of unprocessed elements of f and g:
  c := c + F−h + G−k
```

References

[1] Allen, L.E. *WFF'N PROOF: The Game of Modern Logic.* Autotelic Instructional material Publishers, New Haven, 1972.

[2] Bauer, F.L. and K. Samelson (eds.). *Language Hierarchies and Interfaces.* Springer Verlag Lecture Notes in Computer Science 46, 1976.

[3] Bauer, F.L. and M. Broy (eds.). Software Engineering: an Advanced Course. Springer Verlag Lecture Notes in Computer Science 30, 1975.

[4] ___ and ___ (eds.). *Program Construction.* Springer Verlag Lecture Notes in Computer Science 69, 1979.

[5] Burstall, R. Proving programs as hand simulation with a little induction. *Information Processing 74* (Proceedings of IFIP 74), North-Holland, Amsterdam, 1974, 308-312.

[6] Buxton, J.N., P. Naur, and B. Randell. *Software Engineering*, Petrocelli, 1975. (Report on two NATO Conferences held in Garmisch, Germany (Oct 68) and Rome, Italy (Oct 69)).

[7] Constable, R.L. and M. O'Donnell. *A Programing Logic.* Winthrop Publishers, Cambridge, 1978.

[8] Cook, S.A. Axiomatic and interpretative semantics for an Algol fragment. Computer Science Department, University of Toronto, TR 79, 1975.

[9] Conway, R., and D. Gries. *An Introduction to Programming.* Winthrop Publishers, Cambridge, Mass., 1973 (third edition, 1979).

[10] De Millo, R.A., R.J. Lipton and A.J. Perlis. Social processes and proofs of theorems and programs. *Comm. of the ACM* 22 (May 1979), 271-280.

[11] Dijkstra, E.W. Some meditations on advanced programming. *IFIP 1962*, 535-538.

[12] ___. Go to statement considered harmful. *Comm. ACM* 11 (March

1968), 147-148.

[13] ___. A short introduction to the art of programming. EWD316, Technological University Eindhoven, August 1971.

[14] ___. *Notes on Structured Programming.* In Dahl, O.-J., C.A.R. Hoare and E.W. Dijkstra, *Structured Programming*, Academic Press, New York 1972. (Also appeared a few years earlier in the form of a technical report).

[15] ___. Guarded commands, nondeterminacy and the formal derivation of programs. *Comm. of the ACM* 18 (August 1975), 453-457.

[16] ___. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, 1976.

[17] ___. *Program inversion.* EWD671, Technological University Eindhoven, 1978.

[18] Feijen, W.H.J. A set of programming exercises. WF25, Technological University Eindhoven, July 1979.

[19] Floyd, R. Assigning meaning to programs. In *Mathematical Aspects of Computer Science*, XIX American Mathematical Society (1967), 19-32.

[20] Gentzen, G. Untersuchungen ueber das logische Schliessen. *Math. Zeitschrift* 39 (1935), 176-210, 405-431.

[21] Gries, D. An illustration of current ideas on the derivation of correctness proofs and correct programs. IEEE Trans. Software Eng. 2 (December 1976), 238-244.

[22] ___ (ed.). *Programming Methodology, a Collection of Articles by Members of WG2.3.* Springer Verlag, New York, 1978.

[23] ___ and G. Levin. Assignment and procedure call proof rules. *TOPLAS* 2 (October 1980), 564-579.

[24] ___ and Mills, H. Swapping sections. TR 81-452, Computer Science Dept., Cornell University, January 1981.

[25] Guttag, J.V. and J.J. Horning. The algebraic specification of data types. *Acta Informatica* 10 (1978), 27-52.

[26] Hoare, C.A.R. Quicksort. *Computer Journal* 5 (1962), 10-15.

[27] ___. An axiomatic basis for computer programming. *Comm ACM 12* (October 1969), 576-580, 583.

[28] ___. Procedures and parameters: an axiomatic approach. In *Symposium on Semantics of Programming Languages.* Springer Verlag, New York, 1971, 102-116.

[29] ___. Proof of correctness of data representations. *Acta Informatica* 1 (1972), 271-281.

[30] ___ and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica* 2 (1973), 335-355.

[31] Hunt, J.W. and M.D. McIlroy. An algorithm for differential file comparison. Computer Science Technical Report 41, Bell Labs, Murray Hill, New Jersey, June 1976.

[32] Igarashi, S., R.L. London and D.C. Luckham. Automatic program verification: a logical basis and its implementation. *Acta Informatica* 4 (1975), 145-182.

[33] Liskov, B. and S. Zilles. Programming with abstract data types. *Proc. ACM SIGPLAN Conf. on Very High Level Languages*, SIGPLAN Notices 9 (April 1974), 50-60.

[34] London, R.L., J.V. Guttag, J.J. Horning, B.W. Mitchell and G.J. Popek. Proof rules for the programming language Euclid. *Acta Informatica* 10 (1978), 1-79.

[35] McCarthy, J. A basis for a mathematical theory of computation. *Proc. Western Joint Comp. Conf.*, Los Angeles, May 1961, 225-238, and *Proc. IFIP Congress 1962*, North Holland Publ. Co., Amsterdam, 1963.

[36] Melville R. *Asymptotic Complexity of Iterative Computations.* Ph.D. thesis, Computer Science Department, Cornell University, January 1981.

[37] ___ and D. Gries. Controlled density sorting. *IPL* 10 (July 1980), 169-172.

[38] Misra, J. A technique of algorithm construction on sequences. *IEEE Trans. Software Eng.* 4 (January 1978), 65-69.

[39] Naur, P. et al. Report on ALGOL 60. *Comm. of the ACM* 3 (May 1960), 299-314.

[40] Naur, P. Proofs of algorithms by general snapshots. *BIT 6* (1969), 310-316.

[41] Quine, W.V.O. *Methods of Logic.* Holt, Reinhart and Winston, New York, 1961.

[42] Steel, T.B. (ed.). *Formal Language Description Languages for Computer Programming.* Proc. IFIP Working Conference on Formal Language Description Languages, Vienna 1964, North-Holland, Amsterdam, 1971.

[43] Szabo, M.E. *The Collected Works of Gerhard Gentzen.* North Holland, Amsterdam, 1969.

[44] Wirth, N. Program development by stepwise refinement. *Comm ACM 14* (April 1971), 221-227.

Index

abort, 114
abs, 314
Abstraction, 149
Addition, 314
  identity of, 72
Aho, A.V., 309
Allen, Layman E., 42
Alternative command, 132
  strategy for developing, 174
Ambiguity, 308
An Exercise Attributed to Hamming, 243, 302
and, see Conjunction
and-simplification, 21
Annotated program, 104
Annotation for a loop, 145
Antecedent, 9
Approximating the Square Root, 195, 201, 246, 350
Argument, 152
  final value of, 155
  initial value of, 155
Array, 88
  as a function, 89
  domain of, 89
  two-dimensional, 96
Array of arrays, 96
Array picture, 93
Array Reversal, 214, 302
Array section, 93
Assertion, 2, 100
  output, 100
  result, 100
  placement of, 278
Assignment, simple, 117
  forward rule for, 120
  multiple assignment, 121, 127
  to an array element, 124, 90
Associative laws, 20, 69
  proof of, 48
Associativity of composition, 316
Atomic expression, 67
Axiom, 25

Backus, John, 304
Backus-Naur Form, 304
Balloon theory, 193
Bauer, F.L., 296
BB, 132
Binary relation, 315
Binary Search, 205, 302, 344
Binary tree, 229
BNF, 304
  extensions to, 309
Body, of a procedure, 150-151
Boole, George, 8, 20
Boolean, 8, 66
Bound function, 142
Bound identifier, 76-77
Bound variable substitution, 80,

85
Bounded nondeterminism, 312

Calculus, 25
  propositional calculus, 25
  predicate calculus, 66
Call, of a procedure, 152
  by reference, 158
  by result, 151
  by value, 151
  by value result, 151
cand, 68-70
cand-simplification, 80
Cardinality, of a set, 311
Cartesian product, 315
Case statement, 134
Catenation, 75
  identity of, 75, 333
  of sequences, 312
ceil, 314
Changing a representation, 246
Chebyshev, 83
Checklist for understanding a loop, 145
Chomsky, Noam, 304
Choose, 312
Closing the Curve, 166, 301
Closure, of a relation, 317
  transitive, 317
Code, for a permutation, 270
Code to Perm, 264, 272-273, 303
Coffee Can Problem, 165, 301
Combining pre- and postconditions, 211
Command, 108
  abort, 114
  alternative command, 132
  assignment, multiple, 121, 127
  assignment, simple, 128
  assignment to an array element, 124
  Choose, 312
  deterministic, 111
  guarded command, 131
  iterative command, 139
  nondeterministic, 111
  procedure call, 164
  sequential composition, 114-115
  skip, 114
Command-comment, 99, 279
  indentation of, 279
Common sense and formality, 164
Commutative laws, 20
  proof of, 48
Composition, associativity of, 316
Composition, of relations, 316
Composition, sequential, 114-115
Concatenation, see Catenation
Conclusion, 29
Conjecture, disproving, 15
Conjunct, 9
Conjunction, 9-10
  distributivity of, 110
  identity of, 72
Conjunctive normal form, 27
Consequent, 9
Constable, Robert, 42
Constant proposition, 10
Constant-time algorithm, 321
Contradiction, law of, 20, 70
Contradiction, proof by, 39-41
Controlled Density Sort, 247, 303
cor, 68-70
cor-simplification, 79
Correctness
  partial, 109-110
  total, 110
Counting nodes of a tree, 231
Cubic algorithm, 321
Cut point, 297

Data encapsulation, 235
Data refinement, 235
De Morgan, Augustus, 20
De Morgan's laws, 20, 70
  proof of, 49
Debugging, 5
Decimal to Base B, 215, 302
Decimal to Binary, 215, 302

Declaration, of a procedure, 150
Deduction theorem, 36
Definition, of variables, 283
Deleting a conjunct, 195
Demers, Alan, 302
Depth, of a tree, 236
Derivation, 308
Derived inference rule, 46
  rule of Substitution, 46-47
Determinism, 111
Deterministic command, 111
Difference, of two sets, 311
Different Adjacent Subsequences, 262, 303
Dijkstra, E.W., 295-296, 300-303
disj, 159
Disjoint, pairwise, 159
Disjoint vectors, 159
Disjunct, 9
Disjunction, 9-10
  distributivity of, 111
  identity of, 72
Disjunctive normal form, 27
Distributive laws, 20, 69
  proof of, 48
Distributivity of Conjunction, 110
Distributivity of Disjunction, 111
Divide and Conquer, 226
DO, 138-139
domain, 89, 117
Domain, of an array, 89
Domain, of an expression, 117
Dutch National Flag, 214, 302
Dynamic Programming, 261

Efficient Queues in LISP, 250, 303
Eliminating an Implication, 24
Elimination, rule of, 30
Empty section, 93
Empty set, 310
Empty tree, 229
Enlarging the range of a variable, 206
Equality, 9-10
  law of, 20
equals, see Equality
Equivalence, 19
  laws of, 19-21
Equivalent propositions, 19
Euclid, 301
even, 314
Excluded Middle, law of, 20, 70
Excluded Miracle, law of, 110
Exclusive or, 11
Existential quantification, 71
Exponential algorithm, 321
Exponentiation, 239, 252, 302
Expression, atomic, 67
Expression, domain of, 117

F, 8
Factorial function, 221
Feijen, W.H.J., 264, 302
Fibonacci number, 225
Final value, of a variable, 102
  of an argument, 155
Finding Sums of Squares, 245, 302, 348
Flaw chart, 138, 190-191
  disadvantages of, 275
floor, 314
Floyd, Robert, 297
Formality and common sense, 164
Forward rule for assignment, 120
Four-tuple Sort, 185, 239, 301
Free identifier, 76-77
Function, 318
  bound function, 142
  n-ary function, 319
  of an Identifier, 318
  variant function, 142

gcd, 191, 224-225, 301, 343
Gentzen, Gerhard, 42
Gill, Stanley, 296

Global reference, 38
Grammar, 305
  ambiguous, 308
  sentence of, 305
  unambiguous, 308
Greatest common divisor, see *gcd*
Griffiths, Michael, 301
Guard, 131
Guarded command, 131

Halving an interval, 202
Hamming, R.W., 302
Heading, of a procedure, 282
Hoare, C.A.R., 295, 297-299, 302
Hopcroft, J.E., 309
Horner, W.G., 243
Horner's rule, 242

Identifier, 9
  bound, 76-77
  free, 76-77
  quantified, 71
  quantified, range of, 82
  quantified, type of, 83
  restriction on, 76
Identity element, 72
Identity, law of, 21
Identity relation, 316
Identity, of addition, 72
  of **and**, 72
  of catanation, 75, 333
  of multiplication, 72
  of **or**, 72
IF, 132
Iff, ix
IFIP, 295, 300
imp, see Implication
Implementation of a tree, 230
Implication, 9-10
  elimination of, 24
  law of, 20
Implicit quantification, 83-84
Inclusive or, 11

Indentation, 275
  of command-comments, 279
  of delimiters 279
Inference rule, 25, 30
  bound variable substitution, 85
  derived, 46
  E-E, 85
  E-I, 84
  A-E, 84
  A-I, 84
  =-E, 34, 43
  =-I, 34, 43
  $\Rightarrow$-E, 33, 43
  $\Rightarrow$-I, 36, 43
  $\wedge$-E, 31, 43
  $\wedge$-I, 30-31, 43
  $\vee$-E, 33, 43
  $\vee$-I, 31, 43
  $\neg$-E, 40, 43
  $\neg$-I, 40, 43
Initial value, of a variable, 102
  of an argument, 155
Inorder traversal, 236
inrange, 125
Insertion Sort, 247
Integer, 67, 314
Integer set, 67
Intersection, of two sets, 311
Introduction, rule of, 30
Invariant, 141
Invariant relation, see Invariant
Inversion, 185-186
Inverting Programs, 267
Iteration, 139
Iterative command, 139
  strategy for developing, 181, 187
Ithacating, 31

Justifying Lines, 253, 289-293, 303

Knuth, Donald E., 302

Laws,
and-simplification, 21
proof of, 51
cand-simplification, 70
Associative, 20, 69
Commutative, 20
Contradiction, 20, 70
proof of, 51
cor-simplification, 70
De Morgan's, 20, 70
Distributive, 20, 69
Distributivity of Conjunction, 110
Distributivity of Disjunction, 111
Equality, 20
proof of, 51
Equivalence, 19-21
Excluded Middle, 20, 70
proof of, 50
Excluded Miracle, 110
Identity, 21
Implication, 20
proof of, 51
Monotonicity, 111
Negation, 20
proof of, 50
or-simplification, 21
proof of, 51
Leaf, of a tree, 229
Left subtree, 229
Length, of a sequence, 312
Levin, Gary, 302
Lexicographic ordering, 217
LHS, Left hand side (of an equation)
Linear algorithm, 321
Linear Search, 197, 206, 301
Linear Search Principle, 197, 301
Line Generator, 263
Link Reversal, 215, 302, 346
List of nodes,
inorder, 236
postorder, 236, 347
preorder, 232
log, 314
Logarithm, 314, 321
Logarithmic algorithm, 321
Longest Upsequence, 259, 303
Loop, 139, see Iterative command
annotation for, 145
checklist for understanding, 145
iteration of, 139
lower, 89
lup, 259

make-true, 116
Many-to-one relation, 316
max, 314
Maximum, 301
McCarthy, John, 295
McIlroy, Douglas, 168
Melville, Robert, 303
Meta-theorem, 46
Mills, Harlan, 302
min, 314
Misra, J., 303
mod, 314
Modus ponens, 34
Monotonicity, law of, 111
Multiple assignment, 121, 127
Multiplication, identity of, 72

n-ary function, 319
n-ary relation, 319
Natural deduction system, 29-43
Natural number, 67, 313
Naur, Peter, 297, 304
Negation, 9-10
Negation, law of, 20
Nelson, Edward, 302
Newton, Isaac, 243
Next Higher Permutation, 178, 262, 301, 342
Node Count, 231, 302
Node, of a tree, 229
Non-Crooks, 353, 264
Nondeterminism, 111

bounded, 312
restriction of, 238
unbounded, 312
Nondeterministic command, 111
Nonterminal symbol, 304
Normal form, conjunctive, 27
Normal form, disjunctive, 27
not, see Negation
Null selector, 96
Numerical quantification, 73-74

odd, 314
O'Donnell, Michael, 42
One-to-many relation, 316
One-to-one relation, 316
Onto relation, 316
or, see Disjunction
Order f(n), 321
ordered, 94, 126
Ordered binary tree, 229
Order of execution time, 321
Or, exlusive, 11
inclusive, 11
or-simplification, 21
Outline of a proof, 104
Output assertion, 100

Pairwise disjoint, 159
Paradox, 21
Parameter, 150
var, 158
final value of, 155
initial value of, 155
result, 151
specification for, 150
value, 151
value result, 151
Partial correctness, 109-110
Partial relation, 316
Partition, 214, 302, 345
pdisj, 159
Perfect number, 85
Period, of a Decimal Expansion, 264
perm, 91
Perm, see Permutation
Perm to Code, 263, 270, 303
Permutation, 75
Placement of assertions, 278
Plateau, of an array, 203
Plateau Problem, 203, 206, 301
Postcondition, 100, 109
strongest, 120
postorder, 347
Postorder list, 347
Postorder traversal, 236, 347
Precedence rules, 12, 67
Precondition, 100, 109
weakest, 109
Predicate, 66-87
evaluation of, 67-68
Predicate calculus, 66
Predicate transformer, 109
Predicate weakening, 195
Premise, 29
preorder, 233
Preorder list, 232
Preorder traversal, 232
Prime number, 83
Problems,
An Exercise Attributed to Hamming, 243, 302
Approximating the Square Root, 195, 201, 246, 350
Array Reversal, 214, 302
Binary Search, 205, 302, 344
Closing the Curve, 166, 301
Code to Perm, 264, 272-273, 303
Coffee Can, 301, 165
Controlled Density Sort, 247,303
Decimal to Base B, 215, 302
Decimal to Binary, 215, 302
Different Adjacent Subsequences, 262, 303
Dutch National Flag, 214, 302
Efficient Queues in LISP, 250, 303
Exponentiation, 239, 252, 302

Problems (continued)
Finding Sums of Squares, 245, 302, 348
Four-tuple Sort, 185, 239, 301
gcd, 191, 224-225, 301, 343
Insertion Sort, 247
Justifying Lines, 253, 289-293, 303
Linear Search, 197, 206, 301
Line Generator, 263
Link Reversal, 215, 302, 346
Longest Upsequence, 259, 303
Maximum, 301
Next Higher Permutation, 178, 262, 301, 342
Node Count, 231, 302
Non-Crooks, 353, 264
Partition, 214, 302, 345
Period of a Decimal Expansion, 264
Perm to Code, 263, 270, 303
Plateau Problem, 203, 206, 301
Quicksort, 226, 302
Railroad Shunting Yard, 219
Saddleback Search, 215, 302, 346
Searching a Two-Dimensional Array, 182, 188, 301
Swap, 103, 119
Swapping Equal-Length Sections, 212, 302
Swapping Sections, 222, 302
Tardy Bus, 59
Unique 5-bit Sequences, 262, 303, 352
Welfare Crook, 207, 238, 302
Procedure, 149-162
heading of, 282
argument of a call, 152
body of, 150-151
call of, 152
declaration of, 150
recursive, 217
Production, 304
Program, annotated, 104
Program inversion, 267
Program transformation, 235
Programming-in-the-small, 168
Proof, 163
by contradiction, 39-41
in natural deduction system, 31-32, 41-42
versus test-case analysis, 165
Proof Outline, 104
Proportional to, 321
Proposition, 8-016
as a set of states, 15
constant, 10
equivalent, 19
evaluation of, 11-14
stronger, 16
syntax of, 8-9, 13
weaker, 16
well-defined, 11
Propositional calculus, 25

Quadratic algorithm, 321
Quantification, 71-74
implicit, 83-84
numerical, 73-74
universal, 73
Quantified identifier, 71
range of, 71, 74, 82
type of, 83
Quantifier, existential, 71
Queue, 313
Quicksort, 226, 302
Quine, W.V.O, 42
QWERTY programmer, 170

Railroad Shunting Yard, 219
Randell, Brian, 296
Range of quantified identifier, 71, 74, 82
Record, 92, 98
as a function, 92, 98
Recursive procedure, 221
ref, 159
Refinement, of data, 235

Relation, 315
  binary, 315
  closure of, 317
  composition of, 316
  identity relation, 316
  invariant, see Invariant
  many-to-one, 316
  n-ary, 319
  one-to-many, 316
  one-to-one, 316
  onto, 316
  partial, 316
  total, 316
Rene Descartes, 315
Replacing a constant, 199
Restricting nondeterminism, 238
Result assertion, 100
Result parameter, 151
Rewriting rule, 304
RHS, Right hand side (of an equation)
Right subtree, 229
Role of semicolon, 115
Root, of a tree, 229
Ross, Doug, 296
Routine, 149
Rule of elimination, 30
Rule of inference, 25
Rule of introduction, 30
Rule of Substitution, 22, 26, 46-47
Rule of Transitivity, 23, 26
Rule, rewriting, 304
Rule, see Inference rule

Saddleback Search, 215, 302, 346
Schema, 19, 31, 45
Scholten, Carel, 301
Scope rule, in Algol 60, 38, 78
  in natural deduction system, 38
  in predicates, 78
Searching a Two-Dimensional Array, 182, 188, 301
Section, of an array, 93
  empty section, 93
Seegmueller, Gerhard, 296
SELECT statement, 134
Selector, 96
Semicolon, role of, 115
Sentence, of a grammar, 305
Separation of Concerns, 237
Sequence, 312
  length of, 312
  catenation of, 312
Sequential composition, 114-115
Set, 310
  cardinality of, 311
  difference, 311
  empty, 310
  intersection, 311
  union, 311
Side effects, 119
Simple assignment, 117
Simple variable, 117
Simultaneous substitution, 81
skip, 114
sp, 120
Specification of a parameter, 150
Stack, 313
State, 11
Strategy, for developing a loop, 181, 187
  for developing an alternative command, 174
Strength reduction, 241
Stronger proposition, 16
Strongest postcondition, 120
Strongest proposition, 16
Subscripted variable, 89
Substitution, rule of, 22, 26, 46-47
  simultaneous, 81
  textual, 79-81
  textual, extension to, 128-129
Subtree, 229
Swap, 103, 119
Swapping Equal-Length Sections, 212, 302

Swapping Sections, 222, 302
Symbol of a grammar, 304
  nonterminal, 304
  terminal, 304
Syntax tree, 308

T, 8
Taking an assertion out of a loop, 241
Tardy Bus Problem, 59
Tautology, 14
  relation to *theorem*, 26
Terminal symbol, 304
Textual substitution, 79-81
  extension to, 128-129
Theorem, 25
  relation to tautology, 26
  as a schema, 45
Total correctness, 110
Total relation, 316
Transitive closure, 317
Transitivity, rule of, 23, 26
Traversal, inorder, 236
  postorder, 236, 347
  preorder, 232
Tree, 229
  depth of, 236
  empty, 229
  implementation of, 230
  leaf of, 229
  root of, 229
Truth table, 10, 15
Truth values, 8
Turski, Wlad M., 296
Two-dimensional array, 96

U, 69
Ullman, J.D., 309
Unambiguous grammar, 308
Unbounded nondeterminism, 312
Undefined value, 69
Union, of two sets, 311
Unique 5-bit Sequences, 262, 303, 352
Universal quantification, 73
upper, 89
Upsequence, 259

Value parameter, 151
Value result parameter, 151
Var parameter, 158
Variable, subscripted, 89
  final value of, 102
  initial value of, 102
  definition of, 283
  simple, 117
Variant function, 142

Weakening a predicate, 195
  combining pre- and post-conditions, 211
  deleting a conjunct, 195
  enlarging the range of a variable, 206
  replacing a constant, 199
Weaker proposition, 16
Weakest precondition, 109
Weakest proposition, 16
Welfare Crook, 207, 238, 302
Well-defined proposition, 11
WFF'N PROOF, 28, 42, 59
While-loop, 138
Wilkes, Maurice, 149
Williams, John, 301
Wirth, Niklaus, 296
Woodger, Michael, 296
wp, 108

Texts and Monographs in Computer Science

(continued from page ii)

John V. Guttag and James J. Horning, **Larch: Languages and Tools for Formal Specification**

Eric C.R. Hehner, **A Practical Theory of Programming**

Micha Hofri, **Probabilistic Analysis of Algorithms**

A.J. Kfoury, Robert N. Moll, and Michael A. Arbib, **A Programming Approach to Computability**

Dexter C. Kozen, **The Design and Analysis of Algorithms**

E.V. Krishnamurthy, **Error-Free Polynomial Matrix Computations**

Ming Li and Paul Vitányi, **An Introduction to Kolmogorov Complexity and Its Applications**

David Luckham, **Programming with Specifications: An Introduction to ANNA, A Language for Specifying Ada Programs**

Ernest G. Manes and Michael A. Arbib, **Algebraic Approaches to Program Semantics**

Bhubaneswar Mishra, **Algorithmic Algebra**

Robert N. Moll, Michael A. Arbib, and A.J. Kfoury, **An Introduction to Formal Language Theory**

Anil Nerode and Richard A. Shore, **Logic for Applications**

Helmut A. Partsch, **Specification and Transformation of Programs**

Franco P. Preparata and Michael Ian Shamos, **Computational Geometry: An Introduction**

Brian Randell, Ed., **The Origins of Digital Computers: Selected Papers, 3rd Edition**

Thomas W. Reps and Tim Teitelbaum, **The Synthesizer Generator: A System for Constructing Language-Based Editors**

Thomas W. Reps and Tim Teitelbaum, **The Synthesizer Generator Reference Manual, 3rd Edition**

Arto Salomaa and Matti Soittola, **Automata-Theoretic Aspects of Formal Power Series**

J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg, **Programming with Sets: An Introduction to SETL**

Alan T. Sherman, **VLSI Placement and Routing: The PI Project**

Texts and Monographs in Computer Science

(continued)

Santosh K. Shrivastava, Ed., **Reliable Computer Systems**

Jan L.A. van de Snepscheut, **What Computing Is All About**

William M. Waite and Gerhard Goos, **Compiler Construction**

Niklaus Wirth, **Programming in Modula-2, 4th Edition**

Study Edition

Edward Cohen, **Programming in the 1990s: An Introduction to the Calculation of Programs**