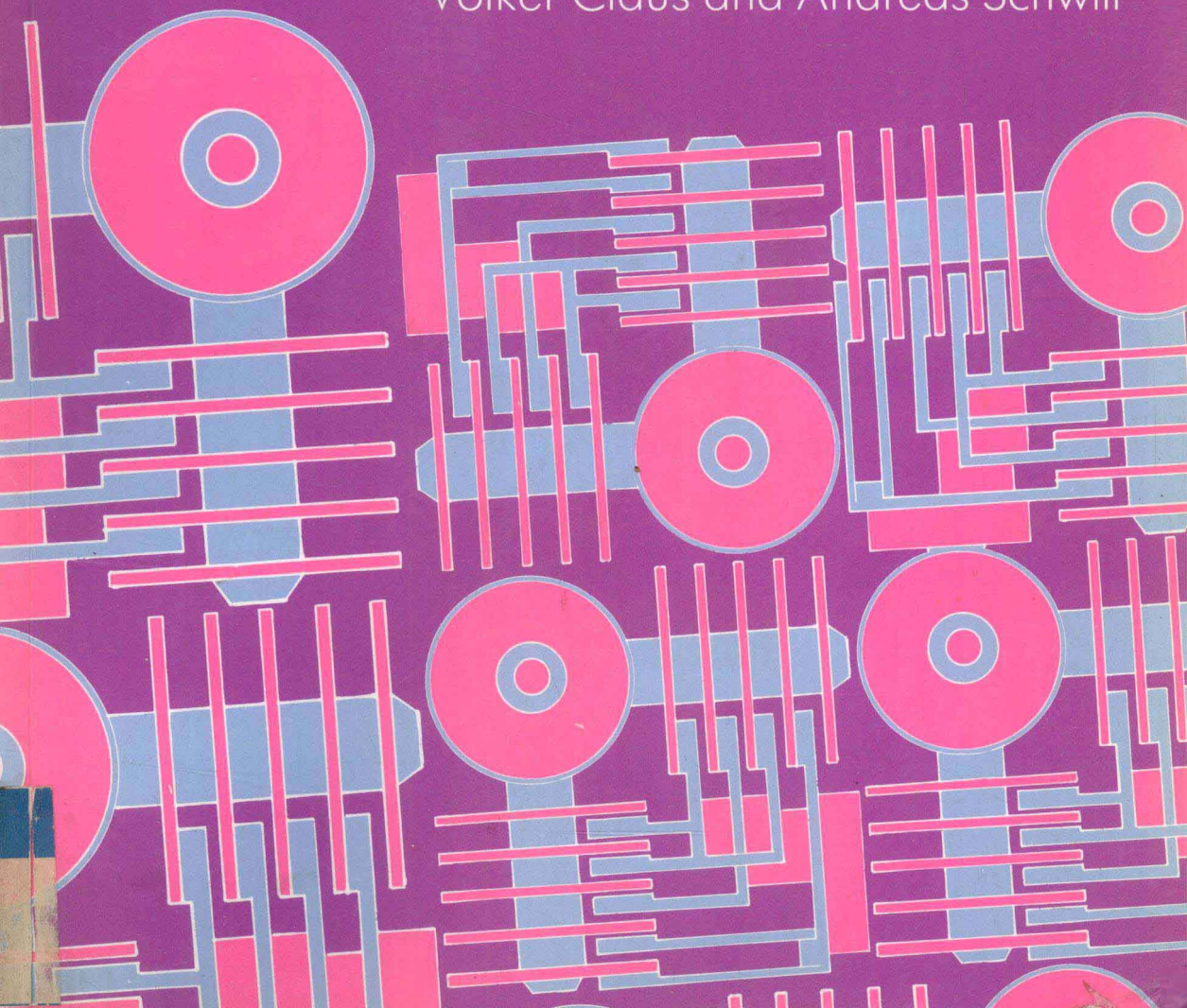
ELLIS HORWOOD BOOKS IN INFORMATION TECHNOLOGY

ENCYCLOPAEDIA OF INFORMATION TECHNOLOGY

Volker Claus and Andreas Schwill



ENCYCLOPAEDIA OF INFORMATION TECHNOLOGY

VOLKER CLAUS Ph.D.
Professor, University of Oldenburg
ANDREAS SCHWILL dipl.Inform.
Research Assistant, University of Oldenburg

Translator
DEREK LEWIS
School of Modern Languages, University of Exeter

ELLIS HORWOOD
NEW YORK LONDON TORONTO SYDNEY TOKYO SINGAPORE

This English edition first published in 1992 by
ELLIS HORWOOD LIMITED
Market Cross House, Cooper Street, Chichester, West Sussex, PO19 1EB, England

A division of
Simon & Schuster International Group

This English edition is translated from the original German edition *Schülerduden=Die Informatik* , published in 1988 by Bibliographisches Institute & F. A. Brockhaus AG, © the copyright holders

Printed and bound in Great Britain
by Hartnolls, Bodmin, Cornwall

British Library Cataloguing in Publication Data

Claus, Volker
Encyclopaedia of information technology. — (Ellis Horwood books in information technology).
1. Computer systems
I. Title. II. Schwill, Andreas.
III. Schülerduden-die Informatik. *English*
004
ISBN 0–13–275728–1

Library of Congress Cataloging-in-Publication Data

Claus, Volker, 1944–
[Schülerduden "Die Informatik". English]
Encyclopaedia of information technology / Volker Claus, Andreas Schweill; translator, Derek Lewis.
p. cm. — (Ellis Horwood books in information technology)
Translation of: Schülerduden "Die Informatik".
ISBN 0–13–275728–1
1. Electronic data processing — Encyclopedias.
2. Information technology — Encyclopedias.
I. Schwill, Andreas. II. Title. III. Series.
QA76.15.C5313 1990
004'.03—dc20 90–44150
CIP

ENCYCLOPAEDIA OF INFORMATION TECHNOLOGY

ELLIS HORWOOD BOOKS IN INFORMATION TECHNOLOGY

General Editor: Professor V. A. J. MALLER, ICL Chair in Computer Systems, Loughborough University of Technology; formerly of Thorn EMI Information Technology Ltd
Consulting Editors: Dr JOHN M. M. PINKERTON, Information Technology Consultant, J & H Pinkerton Associates, Esher, Surrey, and formerly Manager Strategic Requirements, International Computers Limited; and PATRICK HOLLIGAN, Department of Computer Studies, Loughborough University of Technology

C. Baber	SPEECH TECHNOLOGY IN CONTROL ROOM SYSTEMS: A Human Factors Perspective
E. Balagurusamy & J. A. M. Howe	EXPERT SYSTEMS FOR MANAGEMENT AND ENGINEERING
M. Barrett & A. C. Beerel	EXPERT SYSTEMS IN BUSINESS: A Practical Approach
M. Becker, R. Haberfellner & G. Liebetrau	ELECTRONIC DATA PROCESSING IN PRACTICE: A Handbook for Users
A. C. Beerel	EXPERT SYSTEMS: Strategic Implications and Applications
A. C. Beerel	EXPERT SYSTEMS: Real World Applications
K. Bennett	SOFTWARE ENGINEERING ENVIRONMENTS: Research and Practice
D. Berkeley, R. de Hoog & P. Humphreys	SOFTWARE DEVELOPMENT PROJECT MANAGEMENT: Process and Support
A. C. Bradley	OPTICAL STORAGE FOR COMPUTERS: Technology and Applications
P. Brereton	SOFTWARE ENGINEERING ENVIRONMENTS
R. Bright	SMART CARDS: Principles, Practice and Applications
D. Clarke & U. Magnusson-Murray	PRACTICAL MACHINE TRANSLATION
V. Claus & A. Schwill	ENCYCLOPAEDIA OF INFORMATION TECHNOLOGY
D. Cleal & N. O. Heaton	KNOWLEDGE-BASED SYSTEMS: Implications for Human–Computer Interfaces
I. Craig	THE CASSANDRA ARCHITECTURE: Distributed Control in a Blackboard System
T. Daler, *et al.*	SECURITY OF INFORMATION AND DATA
D. Diaper	KNOWLEDGE ELICITATION: Principles, Techniques and Applications
D. Diaper	TASK ANALYSIS FOR HUMAN–COMPUTER INTERACTION
G. I. Doukidis, F. Land & G. Miller	KNOWLEDGE-BASED MANAGEMENT SUPPORT SYSTEMS
P. Duffin	KNOWLEDGE-BASED SYSTEMS: Applications in Administrative Government
C. Ellis	EXPERT KNOWLEDGE AND EXPLANATION: The Knowledge–Language Interface
J. Einbu	A PROGRAM ARCHITECTURE FOR IMPROVED MAINTAINABILITY IN SOFTWARE ENGINEERING
A. Fourcin, G. Harland, W. Barry & V. Hazan	SPEECH INPUT AND OUTPUT ASSESSMENT: Multilingual Methods and Standards
M. Greenwell	KNOWLEDGE ENGINEERING FOR EXPERT SYSTEMS
F. R. Hickman *et al.*	ANALYSIS FOR KNOWLEDGE-BASED SYSTEMS: A Practical Guide to the KADS Methodology
P. Hills	INFORMATION MANAGEMENT SYSTEMS: Implications for the Human–Computer Interface
E. Hollnagel	THE RELIABILITY OF EXPERT SYSTEMS
R. Kerry	INTEGRATING KNOWLEDGE-BASED AND DATABASE MANAGEMENT SYSTEMS
K. Koskimies & J. Paakki	AUTOMATING LANGUAGE IMPLEMENTATION
J. Kriz	KNOWLEDGE-BASED EXPERT SYSTEMS IN INDUSTRY
F. Long	SOFTWARE ENGINEERING ENVIRONMENTS: Volume 3
M. McTear & T. Anderson	UNDERSTANDING KNOWLEDGE ENGINEERING
W. Meyer	EXPERT SYSTEMS IN FACTORY MANAGEMENT: Knowledge-based CIM
U. Pankoke-Babatz	COMPUTER-BASED GROUP COMMUNICATION: The AMIGO Activity Model
J. M. M. Pinkerton	UNDERSTANDING INFORMATION TECHNOLOGY: Basic Terminology and Practice
S. Pollitt	INFORMATION STORAGE AND RETRIEVAL SYSTEMS: Origin, Development and Applications
C.J. Price	KNOWLEDGE ENGINEERING TOOLKITS
S. Ravden & G. Johnson	EVALUATING USABILITY OF HUMAN–COMPUTER INTERFACES: A Practical Method
S. Savory	EXPERT SYSTEMS FOR THE PROFESSIONAL
U. J. Schild	EXPERT SYSTEMS AND CASE LAW
P. E. Slatter	BUILDING EXPERT SYSTEMS: Cognitive Emulation
H. T. Smith, J. Onions & S. Benford	DISTRIBUTED GROUP COMMUNICATION: The AMIGO Information Model
H. M. Sneed	SOFTWARE ENGINEERING MANAGEMENT
M. Stein	BUILDING EXPERT SYSTEMS MODEMS FOR DATA TRANSMISSION
R. Stutely	ADVANCED DESKTOP PUBLISHING: A Practical Guide to Ventura Version 2 and the Professional Extension
J. A. Waterworth	MULTIMEDIA: Technology and Applications
J. A. Waterworth	MULTIMEDIA INTERACTION WITH COMPUTERS
J. A. Waterworth & M. Talbot	SPEECH AND LANGUAGE-BASED COMMUNICATION WITH MACHINES:Towards the Conversational Computer
R. J. Whiddett	THE IMPLEMENTATION OF SMALL COMPUTER SYSTEMS

Foreword

Many subjects in schools and colleges now require a knowledge of information technology (IT) — and the trend is increasing. Above all, the student is expected to develop and formulate his own algorithms for set problems. This means that he has to represent a problem with the help of data structures and construct a set of instructions — the computer algorithm — to arrive at a solution. The algorithm is then converted into a program which is run on a computer and returns an answer to the original problem.

The important thing here is for the student to understand how to proceed from the problem to the algorithm and from the algorithm to the final program. The aim of this book is to shed light on these processes and to provide the student with a useful aid for program design. The book contains numerous examples of practical problems from IT courses, together with their corresponding algorithms and programming solutions (written mostly in PASCAL).

In writing this book, a team of experienced computer scientists has produced descriptions, examples and illustrations of the main concepts of computer science. The emphasis throughout has been on providing definitions which are clear and informative.

The book is directed primarily at secondary school students, teachers and first year undergraduates. Parents who are keen to help their children to understand the basic concepts of computer science and IT may also find the volume useful.

Although the entries are based on established computer science textbooks and school syllabus requirements, we have tried to anticipate the demands which schools are likely to face in the medium term with respect to IT and to take account of current advances in the field.

Mannheim, Spring 1991

The editorial staff

Authors' note

Except where otherwise explicitly stated, the standard conventions for denoting sets of numbers are used in this volume, i.e.

$\mathbb{R}$ is the set of all real numbers;
$\mathbb{Z}$ is the set of all integers;
$\mathbb{Q}$ is the set of all rational numbers;
$\mathbb{N}$ is the set of all natural numbers;

The ◗ symbol denotes a cross-reference to another entry.

A

ABORT

Unplanned termination of a program while it is being executed. A program may be terminated by the user, ◗operator, or the ◗operating system. An operating system abort is usually the result of an ◗error in the program, but it can be caused by a failure in the computer system itself.

ABSTRACT AUTOMATON

A mathematical model of an ◗automaton which is represented in the most generalized way possible. An abstract automaton A is a 7-tuple $A=(I,O,K,\alpha,\omega,\tau,\pi)$ in which

I is a set of input values,
O is a set of output values,
K is a set of *states*,
α: $I\rightarrow K$ is the input function,
ω: $K\rightarrow O$ is the output function,
τ: $K\rightarrow K$ is the transition or change of state function,
π: $K\rightarrow\{0,1\}$ is a predicate (the so-called *halt* predicate).

All functions can be partially defined (◗total function) (Fig. 1). The set E of final states is defined by

$$E=\{k \in K|\pi(k)=1\}.$$

The automaton works as follows. A value is input into a state via α. Then τ is applied repeatedly until the halt predicate is satisfied, i.e. a final state is reached. The final state is converted into an output value via ω. The run time t_A of an abstract automaton A for an input $i \in I$ is given by

$$t_A(i)=\min\{m \in N|\pi(\tau^m(\alpha(i)))=1\}.$$

$t_A(i)$ is thus the number of changes of state which must be made before the halt predicate is satisfied. τ^m represents the mth application of τ.

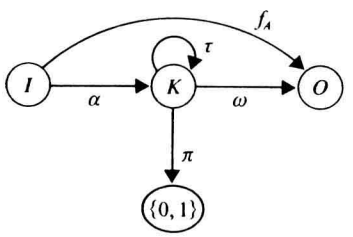


Fig. 1 — Model of an abstract automaton.

The function f_A which is calculated by an abstract automaton A is formally defined by

$$f_A: I\rightarrow O$$

$$f_A(i)=\begin{cases}\omega(\tau^{t_A(i)}(\alpha(i))), \text{ if } t_A(i) \text{ is defined}\\ \text{undefined otherwise}\end{cases}$$

An abstract automaton A is defined so generally that it can calculate any desired function (even non-computable functions (see ◗computable function)).

Example

Let f: $\mathbb{N}\rightarrow\mathbb{N}$ be any (total) function.

An abstract automaton A can be constructed which computes f, i.e. for which $f_A=f$ is true. We have only to use function f within the transition function τ and to define the input and output function as the access to one of a state's components. Let

$$A=(I,O,K,\alpha,\omega,\tau,\pi)$$

where

$$I=O=\mathbb{N} \text{ and } K=\mathbb{N}_0\times\mathbb{N}_0.$$

Let α: $\mathbb{N}\rightarrow\mathbb{N}_0\times\mathbb{N}_0$ be defined by

$$\alpha(n)=(n,0) \text{ for all } n \in \mathbb{N}.$$

ω: $\mathbb{N}_0\times\mathbb{N}_0\rightarrow\mathbb{N}$ is defined by

$$\omega(0,n)=n \text{ for all } n \in \mathbb{N}.$$

The halt or stop predicate

$$\pi: \mathbb{N}_0\times\mathbb{N}_0\rightarrow\{0,1\}$$

is represented by

$$\pi(m,n)=\begin{cases}1, \text{ if } m=0\\ 0 \text{ otherwise}\end{cases}$$

The transition function

$$\tau: \mathbb{N}_0\times\mathbb{N}_0\rightarrow\mathbb{N}_0\times\mathbb{N}_0$$

is defined by

$$\tau(n,0)=(0,f(n)).$$

The run time t_A of A is

$$t_A(n)=\min\{m \in \mathbb{N}|\pi(\tau^m(\alpha(n)))=1\}$$
$$=1$$

for all $n \in \mathbb{N}$.

The function f_A which A computes is given by

$$f_A(n)=\omega(\tau^1(\alpha(n)))=\omega(\tau(n,0))$$
$$=\omega(0,f(n))=f(n)$$

for all $n \in \mathbb{N}$.

For all $n \in \mathbb{N}$, $f_A(n)=f(n)$. Therefore A computes the function f.

By defining I,O,K,α,ω,τ and π, all other models of automata can be derived from the abstract automaton (which is why it is called abstract).

Example
We can derive a finite acceptor (◗finite state automaton) $B=(X,Q,\delta,q_0,F)$ from the abstract automaton if we use the following definition:

$$I := X^*$$
$$K := Q \times X^*$$
$$\pi(q,w) = \begin{cases} 1, \text{ if } q \in F \text{ and } w = \varepsilon \\ 0, \text{ otherwise.} \end{cases}$$

α: $X^* \rightarrow K$ with $\alpha(w)=(q_0,w)$, and τ: $K \rightarrow K$ with $\tau(q,xw) := (\delta(q,x),w)$ for all $x \in X$, $w \in X^*$ and $q \in Q$.

ACCUMULATOR

The accumulator is a ◗register in the ◗central processing unit (CPU). It consists of a number of high speed memories with very fast ◗access times. Before an artihmetic or logical ◗operation is carried out one of the ◗operands is stored in the accumulator. Since the accumulator is involved in each operation, it is not necessary to specify it in the instruction (for a single address machine, ◗instruction format). When the instruction has been performed the accumulator contains the result of the operation, which is available for further processing. This is an advantage where a series of operations produces intermediate results which do not have to be stored in main memory.

ACKERMANN'S FUNCTION

Named after F. W. Ackermann (1896–1962), an example of a ◗computable function which does not exhibit ◗primitive recursion. Ackermann's function

a: $\mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0$

is recursively defined as follows:

$$a(n,m) = \begin{cases} m+1, \text{ if } n = 0 \\ a(n-1,1), \text{ if } m = 0 \\ a(n-1,a(n,m-1)) \text{ otherwise.} \end{cases}$$

Ackermann's function grows very rapidly. An upper bound cannot be set by any primitive recursive function.

Examples

$a(1,1)=3$
$a(2,2)=7$
$a(3,3)=61$
$a(4,2)$ is a number with 19 729 digits
$a(4,4)$ is greater than $10^{10^{10^{19\,000}}}$

ACOUSTIC COUPLER

A device for transmitting data via the telephone network (◗data transmission). Using the same principle as a loudspeaker, an acoustic coupler converts electronic signals from the computer into acoustic wave forms for transmission down the telephone line. To receive signals, a microphone in the coupler converts incoming acoustic wave forms into digital electronic pulses which can be passed to the computer (see Fig. 1).

The telephone receiver is placed in a cradle in the acoustic coupler. The system is simple to use and provides a high degree of protection against noise interference. The advantage of the acoustic coupler over a ◗modem is that no technical alterations are required to the telephone receiver. Owing, however, to the relatively poor transmission quality of the telephone network, the ◗data transmission rate is low. Data exchange (Datex) networks are more suitable for computer communications.

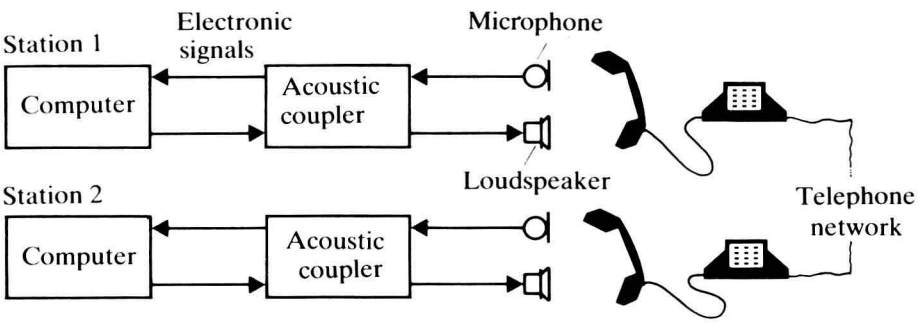


Fig. 1 — Data transmission using an acoustic coupler.

ADA

Named after Augusta Ada Byron, Countess of Lovelace (1815–1852), assistant to Charles Babbage, ADA is an imperative ◗programming language developed by the US Department of Defense. The aim was to replace all programming languages hitherto used by the military with a single, uniform language. ADA combines different concepts from various programming languages. As a result, it is a complex language and requires considerable outlay to develop ◗compilers and programming environments. ADA adopted and extended the strict type concept of the ◗PASCAL programming language, in which every data item is of a particular type (◗data type, ◗typed programming language).

Data types: Like PASCAL, ADA has ◗enumeration types, ◗data types for ◗character, ◗boolean, ◗integer, ◗real (fixed or floating point representation depending on the accuracy required) and subtypes (◗subrange). ADA provides the following ◗data structures: ◗arrays, strings and ◗records (including fixed and variant parts). Among other things, ◗constants, types, ◗variables, ◗procedures and functions can be declared.

Statements: Basic ◗statements include ◗assign (also applies to whole arrays and records), procedure ◗call, a return instruction for exiting from procedures and functions, ◗interrupt, ◗jump, and an exit instruction for terminating ◗loops. Compound statements in ADA are ◗block, ◗if, ◗case, while, and ◗loop, which can all be used in combination.

Packages: Modules, called packages in ADA, can be defined for structuring. A module has two parts (as in ◗MODULA-2). The first part, the package *specification*, contains externally visible constants, variables, types, procedure and function headers and type definitions (where the type name but not the internal structure of the type is visible — so-called private types). The second part, the package *body*, contains the procedures and functions already specified and any additional declarations and statements. These modules can be compiled separately and called from other modules by means of the use statement.

Tasks: In ADA a module which can be executed in ◗parallel (see ◗parallel processing) is called a task. Apart from types, procedures and functions, a specification for a task may include so-called entry call statements. Entry call statements are used to control communication between different task specifications. They can be called from inside these in the same way as procedures. accept statements specify the actions which are to be carried out when an entry call statement is reached. Exactly what happens when a program arrives at an entry call or accept statement depends on the state of the currently running task. The possibilities are as follows:

(1) If a task arrives at an entry call before another task reaches an accept statement for the same entry call, then the task making the call interrupts its processing.
(2) If a task arrives at an accept statement before another task reaches an entry call for it, then the latter task is interrupted.
(3) If one task arrives at an entry call and a second task reaches an accept statement for this call, then everything stops while the second task carries out the actions specified by the accept statement. Both tasks then resume processing. The process by which different tasks communicate or join their activities in this way is called a *rendezvous*.

If several tasks reach entry calls or accept statements and interrupt their processing, they are queued.

ADA contains various statements for controlling interaction between tasks. The most important one is the non-deterministic select statement (◗determinism). This allows execution of a task to be interrupted or another task to be carried out until the appropriate entry call point has been reached. It is also possible to limit the time a task is prepared to wait for an acceptance or to specify alternative actions if the called task cannot accept the call immediately.

Example

The following program listing shows a task specification which manages a table of 100 entries in such a way that various processes can access the table via read and write calls.

```
type index is range 1 .. 100;
task sharedtable is
   -- "element" is a global type
    entry read (i: in index; w: out element);
    entry write (i: in index; w: in element);
end sharedtable;
task body sharedtable
   table: array (index) element;
begin
   loop
      select
         accept read (i: in index; w: out
                                        element) do
            w:= table (i);
         end read;
      or
         accept write (i: in index; w: in element)
                                                do
            w:= table (i);
```

```
        end write;
    end select;
  end loop;
end sharedtable;
```

Generics: In ADA those parts of a program (i.e. procedures, functions, packages and tasks) which share broadly similar features can be declared as generic. When the generic unit is initially declared, such features are specified as parameters. Generic units cannot be directly executed as such. At compile time they must be copied (using the new statement), at which point the actual or current values of the parameters have to be declared. Types and functions can be declared as parameters.

Example
We wish to write a general procedure for sorting an array without reference to the array element type (which may not be known). It is important only to define an ordering relation for this type. In ADA this can be expressed as follows:

```
generic (type elem)
   function "<" (x,y: elem) return elem)
procedure sort (field: in out array (integer range
<>) of elem) is begin
   —Statement of sort procedure
end sort;
procedure intsort is new sort (integer,
                               integer."<");
```

Only in the last line does the new statement declare an executable sort procedure for integers (the ordering relation is defined by integer."<"). Sort procedures for other data types can now be generated in the same way. The generic statement can also be applied to packages and tasks.

Exceptions: A key feature of ADA is error handling. Two approaches are possible. In the first method the programmer may specify in each block the actions to be performed in the event of a ◗run-time error. These might include, for example, outputting a message to the user and terminating the block normally or exiting from the block after passing an error code to either the parent block or the procedure which called it. In the second approach, the programmer himself defines possible errors or exceptional situations. The raise statement generates such an error and passes control to a handler.

Overloading: In many conventional programming languages it is not possible to have identical identifiers within the same block. However, under certain conditions this is allowed in ADA and is called *overloading*.

Example
We can define the following functions in a block, where "index" is a subtype of the class of integers:

```
function "+" (i,j: integer) return integer;
function "+" (i,j: index) return index;
```

The function symbol "+" is overloaded here because it can stand for two functions which are defined differently. The problem of overloading identifiers becomes clear when we consider the following expression:

```
i+3
```

Which of the two above-defined functions is meant here depends on the data type of the arguments. If the variable i has been declared as an integer, the first function is executed. If it is of type index, however, then the second function is meant (assuming that the constant 3 belongs to types integer and index). In complex expressions it may be necessary to analyse the expression in several passes until the declared functions and variables can be assigned to all the identifiers. This highlights the advantages and disadvantages of overloading. On the one hand, it enables many problems to be formulated more simply (e.g. a sort procedure can use the same identifiers for both integers and text). On the other, the complexity of assigning identifiers to declarations can result in the ADA compiler assigning functions which the programmer never intended.

In ADA specifications for particular hardware can be included. Thus the programmer can tell the ADA compiler how to implement particular data types (i.e. the number of bytes to reserve, etc.). Furthermore he can define the hardware configuration on which the program is to run and integrate sections of machine code into an ADA program.

A complete ADA program consists of a number of modules (packages and tasks), each of which can be compiled separately. Altering and subsequently compiling an individual module may require recompilation of all the other modules which are dependent on it. For this reason all ADA modules (source and ◗object programs) are managed in a single program library.

ADA was developed with the object of replacing a large number of different programming languages by a single standard language. Combining numerous different methods and approaches, it is an exceedingly complicated language and its compilers are large and very complex. Users of ADA and even some programmers find it hard to grasp its ◗semantics. For this reason some experts tend not to recommend ADA, especially for applications where data security is paramount.

ADDER

An essential component of a ◗central processing unit (CPU) which adds two or more ◗operands. Since all four basic arithmetical operations can be reduced to addition, the adder can also be used for subtraction, multiplication and division. For implementations of adders see ◗serial adder, ◗parallel adder, and ◗von Neumann adder.

ADDRESS

A number or a character sequence used to identify a storage location in a computer (◗memory). A single location or a set of locations may be addressed. Addresses are integers, from 0 to 2^n-1 for a number n, where n is usually either 16 or 32. To access the contents of a memory location, an ◗address field in a ◗machine language ◗instruction specifies the address of the location. An instruction usually contains one, two or three addresses (referred to as a one, two or three address system). The *address format* of an instruction determines the number and meaning of the addresses. The *address range* is the set of all possible addresses and depends on the processor type, memory size and ◗operating system.

If the computer is constructed in such a way that the address of an object specified in the address field differs from the actual physical address of the memory location, then the operating system must calculate the real value using additional information. Typical ◗addressing methods requiring a calculation of this kind are indirect, indexed, relative, symbolic and virtual addressing.

ADDRESS FIELD

Part of an ◗instruction in ◗machine language or ◗assembler which contains information about the ◗addresses of ◗operands (◗instruction format). The address field may contain several addresses.

ADDRESS FUNCTION

When it accesses the contents of locations in a computer ◗memory, a program often does not know the absolute ◗addresses as such. Instead, it uses relative addressing (◗relocatable program, ◗addressing methods). The same principle applies to arrays. To address an element of an ◗array, the absolute address in the memory must first be established via the index values and the location of the whole array in the memory. The address function then works out the absolute address.

(1) A ◗machine program using relative addressing is loaded into the computer's work space (◗loader). It is loaded to a fixed address, called the load address or LA (◗start address). Every relative address (RA) which occurs in the program must be converted into an absolute address (AA). This is done according to the following formula or address function:

AA :=LA+RA

(2) In a virtual memory system (◗memory management), the contents of areas of memory (called pages) are removed while the program is being executed. Later these may be reloaded to another point in the memory if required. For these purposes the program must be relocatable. To access data in subsequently loaded pages, the same address function is used as in (1), where LA is the actual address at which a page is loaded.

(3) Actual addresses also have to be calculated in order to access the contents of ◗data structures (e.g. arrays or ◗records). What address function is used depends on the type of data structure. If, for example, an array is declared (◗declaration) as

var F: array [u_1 .. o_1, u_2 .. o_2] of integer;

then the actual address of an element $F[i,j]$ in the array is as follows (where a_0 is the absolute start address of the array in memory and a data item of type integer is assumed to occupy two storage locations):

Address $(F[i,j])=$
$a_0+2\cdot((o_2-u_2+1)\cdot(i-u_1)+(j-u_2))$

This address function assumes that the two-dimensional array F is accessed in memory a line at a time. First of all the elements of the first line are accessed (first index$=u_1$), then the elements of the second line (first index$=u_1+1$), and so on. We can specify a general formula for this. Let us assume an n-dimensional array

var F: array [u_1 .. o_1, u_2 .. o_2, ... , u_n .. o_n] of type

whose elements are of data type type. Each element occupies r memory locations and a_0 is the absolute start address of the array F in memory. The absolute address of the memory location for the beginning of the element $F[k_1,k_2, \ldots, k_n]$ is given by the address function f:

$$f(k_1,k_2, \ldots, k_n) = a_0 + r\cdot\sum_{i=1}^{n} d_i\cdot(k_i - u_i),$$

where the so-called edge lengths d_i are given by

$$d_i = \prod_{j=i+1}^{n} (o_j - u_j + 1)$$

These are easily worked out in order:

$d_n = 1$ and $d_{j-1} = d_j \cdot (o_j - u_j + 1)$
for $j = n, n-1, \ldots, 2$

The address function can be simplified somewhat by taking all values which are independent of the indices k and combining them into a so-called "reduced start address":

$$a_{red} = a_0 - r \cdot \sum_{i=1}^{n} d_i \cdot u_i$$

This gives

$$f(k_1, \ldots, k_n) = a_{red} + r \cdot \sum_{i=1}^{n} d_i \cdot k_i$$

In this form f is also referred to as the *memory image function*.

For calculating the addresses of n-dimensional arrays we use the following vector:

$(a_{red}, n, r, d_1, d_2, \ldots, d_n)$.

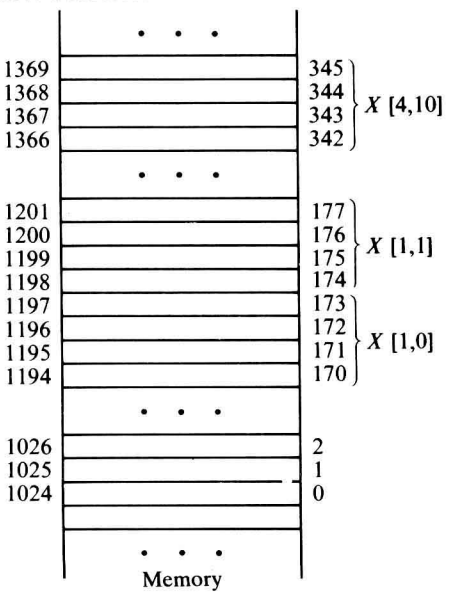
To check for indexing errors (i.e. if k_i should be outside the range u_i to o_i), all upper and lower limits for u_i and o_i respectively should be stored and checked against.

Example

We wish to access elements $X[3,5]$ of the array X. X has been declared as

array [1 .. 4, 0 .. 10] of real.

Let $r = 4$ and X be assigned the relative start address 170. The load address (LA) is 1024. The following table shows how things are organized in the computer's memory; on the left are the absolute addresses of the elements in the array, on the right the relative addresses.



The absolute start address a_0 of X is arrived at as follows:

$a_0 = 1024 + 170$
$= 1194$

The edge lengths are given by $d_2 = 1$ and $d_1 = 11$. With the reduced start address

$a_{red} = a_0 - r \cdot (d_1 \cdot u_1 + d_2 \cdot u_2)$
$= 1194 - 4 \cdot (11 \cdot 1 + 1 \cdot 0)$
$= 1150$

we can calculate the absolute address of $X[3,5]$:

address($X[3,5]$) $= a_{red} + r \cdot (d_1 \cdot 3 + d_2 \cdot 5)$
$= 1150 + 4 \cdot (33 + 5)$
$= 1302$

ADDRESS REGISTER

A ◗register which is reserved for holding an ◗address. An address register (e.g. the base register and the index register) is required for certain ◗addressing methods. The main ◗memory of a computer also has a special address register. Called a *memory address register*, this is used to access a single memory location during a cycle.

ADDRESSING METHODS

An addressing method specifies a way of establishing a physical address. Many ◗machine program instructions include an ◗address field. The address field holds the ◗address of a memory location which contains the ◗operands or into which data values are entered. The information contained in the address field may not be identical to the physical address but is used to calculate it during processing.

Figs 1–5 illustrate different addressing methods using a memory store of seven memory locations, addressed from 1 to 7.

In *absolute* (or *direct*) *addressing* the address field directly specifies the actual address of the operand.

Example

Load register with address 3 (Fig. 1).

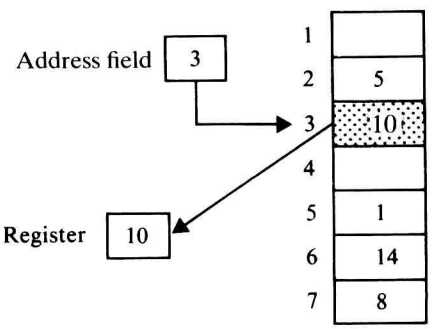


Fig. 1 — Absolute or direct addressing.

In *indirect addressing* the address field specifies an address which in turn contains another address. This second address is the memory location containing the operand.

Example
Load register indirectly with address 2 (Fig. 2).

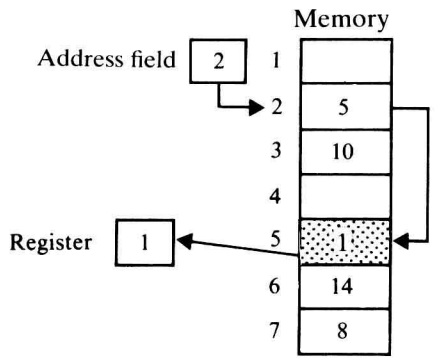


Fig. 2 — Indirect addressing.

In *indexed addressing* the address of the operand is arrived at by adding the address specified in the address field to the content of a special *index register*.

Example
An address register contains the address for memory location 4. To this is added the content of the index register (2). This allows the content of memory location 6 to be accessed and loaded into the first register (Fig. 3).

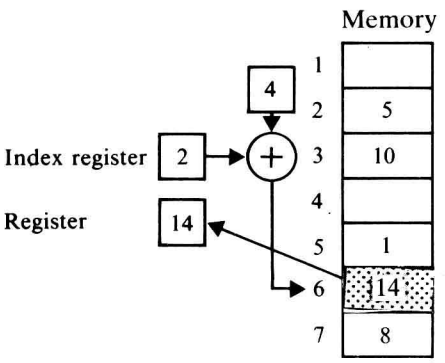


Fig. 3 — Indexed addressing.

Indexed addressing is particularly useful for rapidly accessing consecutive memory locations.

In *relative addressing* a program determines the address of each operand by adding the content of a *base register* (the *base address*) to that of the address field (referred to as the *distance address*, *displacement* or *offset*). The principle is similar to indexed addressing. The base address is usually deposited in the base register at the start of the program or subroutine. It is then automatically accessed for each address calculation, without being explicitly mentioned in the program instruction.

Examples
(1) To load a register with the content of memory location 5 (Fig. 4), the memory location contained in the address field (3) is added to the content of the base register (2).

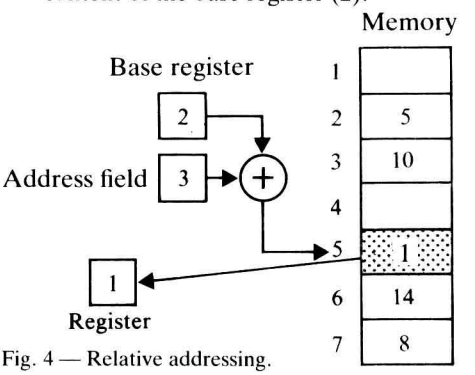


Fig. 4 — Relative addressing.

(2) To load a register with the content of memory location 7, the content of the base register, address field and index register are added (Fig. 5).

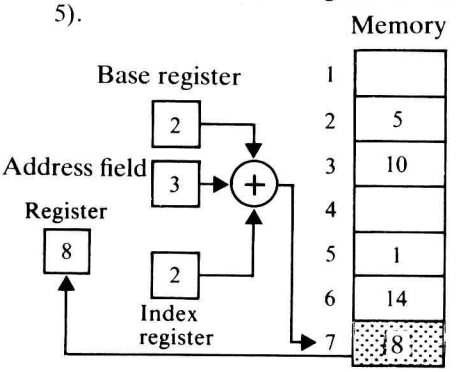


Fig. 5 — Relative addressing.

Relative addressing is important for ▸relocatable programs.

In *symbolic addressing* the programmer freely assigns a name or label to a memory location. This is usually a mnemonic (e.g. POC for Post Office Code). The program ▸compiler substitutes absolute addresses for the symbolic ones. Symbolic addressing is used mainly in ▸assembler programming.

Example
The memory location whose address is 4 has the symbolic label POC (Fig. 6). The assembler translates the instruction, "load register with the content of POC" into "load register with the content of memory location 4" (Fig. 7).

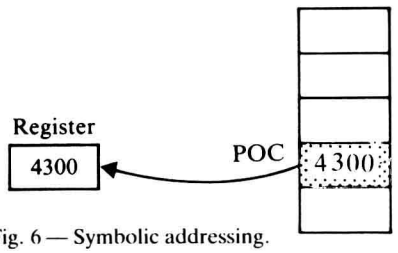


Fig. 6 — Symbolic addressing.

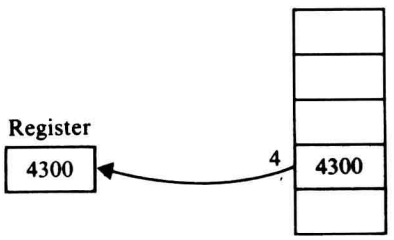
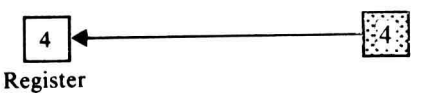


Fig. 7 — Translation of a symbolic address into an absolute address.

A virtual memory system (◗memory management) appears to the user as a single large addressable area in ◗main memory. In fact this (virtual) area is much larger than the physical main memory of the machine. Programs and data are accessed via a *virtual addressing* system. The actual or physical addresses which correspond to the virtual addresses are assigned by the operating system, which loads data in and out of an external memory store as it is needed.

In *immediate addressing* the address in the address field is the operand itself (*direct operand*). The memory itself is not accessed.

Example
Load register with address 4.



For addressing purposes, the term *label* may refer to another point in the program, a component of the computer or a peripheral device.

AIKEN CODE

The Aiken code is a tetradic code, i.e. made up of four elements (◗BCD code). A tetrad (◗byte) is assigned to each of the decimal numbers between 0 and 9 according to the following table.

Decimal number	Binary code	Decimal number	Binary code
0	0000	5	1011
1	0001	6	1100
2	0010	7	1101
3	0011	8	1110
4	0100	9	1111

Example
The decimal number 197 is represented as 0001 1111 1101 (four binary digits for each decimal number).

In the Aiken code the first position from the right has the place value 1, the second 2, the third 4 and the fourth 2. For this reason the Aiken code is also referred to as *2–4–2–1* code.

Example

$$7=1101=1\cdot2+1\cdot4+0\cdot2+1\cdot1$$

Like ◗excess-3 code, Aiken code is a ◗complementary code.

ALGOL 60

An abbreviation for Algorithmic Language, ALGOL 60 is an imperative ◗programming language which was developed in the late 1950s and whose ◗syntax was the first to be formally defined (in ◗Backus–Naur form). ALGOL 60 was the first programming language to support ◗procedures (including recursion), ◗blocks and high level ◗control structures. ALGOL 60 is the precursor of numerous imperative languages such as ◗PASCAL, ◗SIMULA and ◗ALGOL 68. It is influenced by ◗Lambda calculus and ◗assembler languages.

Every ALGOL 60 program is a block containing ◗declarations and ◗instructions. Basic ◗data types are ◗boolean, ◗integer, ◗real and ◗label. The only ◗data structure allowed is the (multidimensional) ◗array. Procedures and functions can be defined in the declaration part of each block. It has basic instructions for assigning values (◗assign), calling procedures (◗call) and jumping to other parts of the program (◗jump). Control statements are ◗if, a special form of ◗case, blocks, while and count ◗loops. Conditional statements can also be used inside ordinary statements. Thus the statement

(1) a: = if z < 0 then 0 else 1

is equivalent in ALGOL 60 to

```
if z < 0 then a: = 0 else a: = 1
```

Similarly

(2) goto if a = 1 then end
else beginning

means the same as

```
if a = 1 then goto end else
    goto beginning
```

Conditional statements can also be used in declarations, e.g.

```
begin
    real array m[if a = 1 then − 10
                          else 10:100];
end
```

Depending on the value of a, we declare either an array

```
real array m[ − 10:100]
```

or an array

```
real array m[10:100]
```

Many ALGOL 60 structures have been adopted by other programming languages, in either a similar or an extended form. Languages which are fundamentally similar to ALGOL 60 are hence referred to as *ALGOL-type languages*. ALGOL 60 has led to intensive research into techniques of compilation (◗compiler). It has been used in implementing ◗run-time systems, orienting compilers to the syntax of programming languages and in various optimization techniques.

ALGOL 68

An imperative ◗programming language which is an advanced and logical development of ◗ALGOL 60. ALGOL 68 has its own basic design philosophy, the so-called *orthogonal* principle. According to this the language consists of a minimal number of universal elements which can be used in any desired combination.

(1) A minimal number of basic elements is available for ◗objects. In ALGOL 68 these elements are the standard data types ◗boolean, character, ◗integer and ◗real, as well as syntactically defined character strings.

(2) From these basic elements, special constructors, or *mode-makers*, can be used to build up more complex data objects. These so-called *object structures* (referred to as *modes* in ALGOL 68) may be given a name and used to construct further data objects. This constant generation of new object structures by means of constructors can be repeated without limit for as long or as often as required. The five most important constructors for object structures in ALGOL 68 are

[]	for constructing a 1-dimensional array (◗array)
struct	for constructing ◗records
union	for representing the combination of several object structures
ref	for constructing an "address" (setting up a reference or ◗pointer)
proc	for producing a function or ◗procedure

(3) The number of possible actions is kept to a minimum. The first of these are simple operations defined on basic elements, such as addition, division, equality, <, ≦, >, rounding, absolute value, logical OR, etc. The second group includes casting operations when passing from one basic element to another (e.g. integer to real). Thirdly, we can distinguish operations which are associated with constructors. These include searching for an element in a structure (*selection*, e.g. of the *i*th element in a array) or passing from a pointer object to the object being pointed to (*dereferencing*). Fourthly, we can include the generation of ◗variables. Whenever a variable is declared in a program, a suitable memory location (address) is allocated to it (or "generated"). The address is either assigned with respect to the correct ◗block (stack-oriented memory management declaration by loc) or it is not (declaration by heap generator). The fifth category of basic action includes the ◗assign, ◗call and ◗jump statements.

(4) Just as object structures can be built up freely from primitive elements, so can action constructors be used to assemble complex actions from the basic ones. This corresponds to the ◗expressions and ◗control structures of other programming languages. Constructors of this type in ALGOL 68 are as follows:

— The "go-on" symbol (semicolon), where the action after the semicolon begins once the action before it is completed;
— concurrent action constructor (represented by a series of actions separated by commas and enclosed by begin ... end; see entry under ◗concurrency);
— the if and case instruction;
— the while and count ◗loops;
— exit (leave a structure and return a value).

Apart from the basic orthogonal design principle, ALGOL 68 also has a number of other important features. Examples are as follows.

— The concept of *referencing*, i.e. of referring to one object structure by another. This is embodied in the constructor ref.
— The concept of *complete typing*. Each value or element which occurs must be a member of a particular object structure (i.e. mode).
— The concept of *complete description*. This means that, for every syntactically correct ALGOL 68 program, it is clear how it functions and executes, i.e. a full ◗semantic description is also implied.

The orthogonal structure and other principles of program design are reflected in many programming languages which emerged after 1965 (such as ◗PASCAL and ◗ELAN). In the current state of the art some features of ALGOL 68 (e.g. the value returned by a function is itself a function) can be implemented with an ◗interpreter but not a ◗compiler — at least, not according to existing criteria of ◗efficiency. Only a small number of compilers for certain dialects of ALGOL 68 are currently available, which limits the computers on which programs can be run. Although the revised Report of 1973 marked the end of the development of ALGOL 68, it has had a lasting influence on the evolution of other imperative programming languages.

ALGORITHM

An algorithm is a precisely formulated set of instructions which can be carried out by a mechanical or electronic device. Examples of algorithms are instructions for adding, subtracting or multiplying numbers, ◗Euclid's algorithm for calculating the greatest common divisor (gcd), and all procedures formulated in a ◗programming language. Cooking recipes, model-making instructions, musical scores, rules for playing games and a number of procedures and conventions encountered in everyday life have algorithmic character. They are rarely explicitly formulated, however, and often have to be interpreted by whoever is carrying them out.

A computer algorithm specifies the step-by-step process by which input data is converted into output data. Expressed formally, an algorithm is a mapping, f: $I \rightarrow O$, of the set of input data I into the set of output data O. Historically, some attempts were made to define exactly the mapping operations which algorithms represent. However, the set of primitive recursive functions (◗primitive recursion) proved inadequate for such a definition (◗Ackermann's function), with the result that partially recursive functions were introduced instead.

An algorithm can be described by a machine which is mathematically precisely defined and which carries out (executes) the steps specified in the algorithm. The simplest models for such machines are ◗Turing and ◗register machines. Such models are valid if the machines they represent can in principle be physically constructed. They include mathematical models of existing computers which are assumed to have unlimited ◗memory capacity. However, it is not necessary actually to build the machine. It is enough to define its basic ◗instruction set and ◗control structures, i.e. the permissible combinations of the basic instructions; we also specify how and where the device will store data. In other words, we define, not a *real*, but a *virtual* machine. This means that we know either that it is possible in theory to build such a device or that its operations can be simulated by another machine. Virtual machines are usually described using imperative programming languages, which is why programs are always algorithms. We refer to an algorithm formulated in such a way that it can be executed by a computer as a *program*. For a particular machine an ◗interpreter or a ◗compiler translates the program into the computer's own instruction set; the result is an executable program.

The notion of an algorithm is very important in computer science. It may be understood in a number of different ways, e.g. a 'program', a 'computable function' or a 'solution to a set of logic formulae'. In principle an algorithm is anything which a mahcine is able to process. In this sense computers are said to execute algorithms. If a problem cannot be solved algorithmically (◗decidability) then it cannot be solved by a computer or any other machine.

We illustrate the various ways of describing or representing algorithms using the factorial function as an example.

$$f(n) = n! = 1 \cdot 2 \ldots \cdot n$$

It is assumed that positive numbers are input into the programs.

(a) The algorithm represented as a ◗recursion model:

$$f(0) = 1$$
$$f(x + 1) = (x + 1) \cdot f(x)$$

(b) The algorithm as a ◗BASIC program:

```
10 INPUT N
20 J = 1
30 IF N = 0 THEN 70
40 J = J * N
50 N = N - 1
60 GOTO 30
70 PRINT J
80 END
```