Algorithms and Data Structures

Translator's note. *This book was translated into Russian in 2009 for specific teaching purposes. Along the way, Pascal-to-Modula-2-to-Oberon conversion typos were corrected and some changes to programs were made. The changes (localized in sects. 1 and 3) were agreed upon with the author in April, 2009. Their purpose was to facilitate verification of the program examples that are now in perfect running order. Most notably, section 1.9 now uses the Dijkstra loop introduced in Oberon-07 (see Appendix C).*

This book can be downloaded from the author's site:

http://www.inf.ethz.ch/personal/wirth/books/AlgorithmE1/AD2012.pdf

The program examples can be downloaded from the site that promulgates Oberons as a unique foundation to teach kids from age 11 all the way up through the university-level compiler construction and software architecture courses:

http://www.inr.ac.ru/~info21/ADen/

where the most recently corrected version may be available.

Please send typos and bugs to: info21@inr.ac.ru

Thanks are due to Wojtek Skulski, Nicholas J. Schwartz and — the longest list of typos by far — Helmut Zinn.

—Fyodor Tkachov, Moscow, 2012-02-18
Last update 2012-03-06

Table of Contents

Preface

In recent years the subject of computer programming has been recognized as a discipline whose mastery is fundamental and crucial to the success of many engineering projects and which is amenable to scientific treatement and presentation. It has advanced from a craft to an academic discipline. The initial outstanding contributions toward this development were made by E.W. Dijkstra and C.A.R. Hoare. Dijkstra's *Notes on Structured Programming* [1] opened a new view of programming as a scientific subject and intellectual challenge, and it coined the title for a "revolution" in programming. Hoare's *Axiomatic Basis of Computer Programming* [2] showed in a lucid manner that programs are amenable to an exacting analysis based on mathematical reasoning. Both these papers argue convincingly that many programming errors can be prevented by making programmers aware of the methods and techniques which they hitherto applied intuitively and often unconsciously. These papers focused their attention on the aspects of composition and analysis of programs, or more explicitly, on the structure of algorithms represented by program texts. Yet, it is abundantly clear that a systematic and scientific approach to program construction primarily has a bearing in the case of large, complex programs which involve complicated sets of data. Hence, a methodology of programming is also bound to include all aspects of data structuring. Programs, after all, are concrete formulations of abstract algorithms based on particular representations and structures of data. An outstanding contribution to bring order into the bewildering variety of terminology and concepts on data structures was made by Hoare through his *Notes on Data Structuring* [3]. It made clear that decisions about structuring data cannot be made without knowledge of the algorithms applied to the data and that, vice versa, the structure and choice of algorithms often depend strongly on the structure of the underlying data. In short, the subjects of program composition and data structures are inseparably interwined.

Yet, this book starts with a chapter on data structure for two reasons. First, one has an intuitive feeling that data precede algorithms: you must have some objects before you can perform operations on them. Second, and this is the more immediate reason, this book assumes that the reader is familiar with the basic notions of computer programming. Traditionally and sensibly, however, introductory programming courses concentrate on algorithms operating on relatively simple structures of data. Hence, an introductory chapter on data structures seems appropriate.

Throughout the book, and particularly in Chap. 1, we follow the theory and terminology expounded by Hoare and realized in the programming language *Pascal* [4]. The essence of this theory is that data in the first instance represent abstractions of real phenomena and are preferably formulated as abstract structures not necessarily realized in common programming languages. In the process of program construction the data representation is gradually refined in step with the refinement of the algorithm to comply more and more with the constraints imposed by an available programming system [5]. We therefore postulate a number of basic building principles of data structures, called the fundamental structures. It is most important that they are constructs that are known to be quite easily implementable on actual computers, for only in this case can they be considered the true elements of an actual data representation, as the molecules emerging from the final step of refinements of the data description. They are the record, the array (with fixed size), and the set. Not surprisingly, these basic building principles correspond to mathematical notions that are fundamental as well.

A cornerstone of this theory of data structures is the distinction between fundamental and "advanced" structures. The former are the molecules themselves built out of atoms that are the components of the latter. Variables of a fundamental structure change only their value, but never their structure and never the set of values they can assume. As a consequence, the size of the store they occupy remains constant.

"Advanced" structures, however, are characterized by their change of value and structure during the execution of a program. More sophisticated techniques are therefore needed for their implementation. The sequence appears as a hybrid in this classification. It certainly varies its length; but that change in structure is of a trivial nature. Since the sequence plays a truly fundamental role in practically all computer systems, its treatment is included in Chap. 1.

The second chapter treats sorting algorithms. It displays a variety of different methods, all serving the same purpose. Mathematical analysis of some of these algorithms shows the advantages and disadvantages of the methods, and it makes the programmer aware of the importance of analysis in the choice of good solutions for a given problem. The partitioning into methods for sorting arrays and methods for sorting files (often called internal and external sorting) exhibits the crucial influence of data representation on the choice of applicable algorithms and on their complexity. The space allocated to sorting would not be so large were it not for the fact that sorting constitutes an ideal vehicle for illustrating so many principles of programming and situations occurring in most other applications. It often seems that one could compose an entire programming course by choosing examples from sorting only.

Another topic that is usually omitted in introductory programming courses but one that plays an important role in the conception of many algorithmic solutions is recursion. Therefore, the third chapter is devoted to recursive algorithms. Recursion is shown to be a generalization of repetition (iteration), and as such it is an important and powerful concept in programming. In many programming tutorials, it is unfortunately exemplified by cases in which simple iteration would suffice. Instead, Chap. 3 concentrates on several examples of problems in which recursion allows for a most natural formulation of a solution, whereas use of iteration would lead to obscure and cumbersome programs. The class of backtracking algorithms emerges as an ideal application of recursion, but the most obvious candidates for the use of recursion are algorithms operating on data whose structure is defined recursively. These cases are treated in the last two chapters, for which the third chapter provides a welcome background.

Chapter 4 deals with dynamic data structures, i.e., with data that change their structure during the execution of the program. It is shown that the recursive data structures are an important subclass of the dynamic structures commonly used. Although a recursive definition is both natural and possible in these cases, it is usually not used in practice. Instead, the mechanism used in its implementation is made evident to the programmer by forcing him to use explicit reference or pointer variables. This book follows this technique and reflects the present state of the art: Chapter 4 is devoted to programming with pointers, to lists, trees and to examples involving even more complicated meshes of data. It presents what is often (and somewhat inappropriately) called list processing. A fair amount of space is devoted to tree organizations, and in particular to search trees. The chapter ends with a presentation of scatter tables, also called "hash" codes, which are often preferred to search trees. This provides the possibility of comparing two fundamentally different techniques for a frequently encountered application.

Programming is a constructive activity. How can a constructive, inventive activity be taught? One method is to crystallize elementary composition priciples out many cases and exhibit them in a systematic manner. But programming is a field of vast variety often involving complex intellectual activities. The belief that it could ever be condensed into a sort of pure recipe teaching is mistaken. What remains in our arsenal of teaching methods is the careful selection and presentation of master examples. Naturally, we should not believe that every person is capable of gaining equally much from the study of examples. It is the characteristic of this approach that much is left to the student, to his diligence and intuition. This is particularly true of the relatively involved and long example programs. Their inclusion in this book is not accidental. Longer programs are the prevalent case in practice, and they are much more suitable for exhibiting that elusive but essential ingredient called style and orderly structure. They are also meant to serve as exercises in the art of program reading, which too often is neglected in favor of program writing. This is a primary motivation behind the inclusion of larger programs as examples in their entirety. The

reader is led through a gradual development of the program; he is given various snapshots in the evolution of a program, whereby this development becomes manifest as a stepwise refinement of the details. I consider it essential that programs are shown in final form with sufficient attention to details, for in programming, the devil hides in the details. Although the mere presentation of an algorithm's principle and its mathematical analysis may be stimulating and challenging to the academic mind, it seems dishonest to the engineering practitioner. I have therefore strictly adhered to the rule of presenting the final programs in a language in which they can actually be run on a computer.

Of course, this raises the problem of finding a form which at the same time is both machine executable and sufficiently machine independent to be included in such a text. In this respect, neither widely used languages nor abstract notations proved to be adequate. The language Pascal provides an appropriate compromise; it had been developed with exactly this aim in mind, and it is therefore used throughout this book. The programs can easily be understood by programmers who are familiar with some other high-level language, such as ALGOL 60 or PL/1, because it is easy to understand the Pascal notation while proceeding through the text. However, this not to say that some proparation would not be beneficial. The book *Systematic Programming* [6] provides an ideal background because it is also based on the Pascal notation. The present book was, however, not intended as a manual on the language Pascal; there exist more appropriate texts for this purpose [7].

This book is a condensation and at the same time an elaboration of several courses on programming taught at the Federal Institute of Technology (ETH) at Zürich. I owe many ideas and views expressed in this book to discussions with my collaborators at ETH. In particular, I wish to thank Mr. H. Sandmayr for his careful reading of the manuscript, and Miss Heidi Theiler and my wife for their care and patience in typing the text. I should also like to mention the stimulating influence provided by meetings of the Working Groups 2.1 and 2.3 of IFIP, and particularly the many memorable arguments I had on these occasions with E. W. Dijkstra and C.A.R. Hoare. Last but not least, ETH generously provided the environment and the computing facilities without which the preparation of this text would have been impossible.

Zürich, Aug. 1975 N. Wirth

[1] E.W. Dijkstra, in: O.-J. Dahl, E.W. Dijkstra, C.A.R. Hoare. Structured Programming. F. Genuys, Ed., New York, Academic Press, 1972, pp. 1-82.

[2] C.A.R. Hoare. *Comm. ACM*, 12, No. 10 (1969), 576-83.

[3] C.A.R. Hoare, in Structured Programming [1], cc. 83-174.

[4] N. Wirth. The Programming Language Pascal. *Acta Informatica*, 1, No. 1 (1971), 35-63.

[5] N. Wirth. Program Development by Stepwise Refinement. *Comm. ACM*, 14, No. 4 (1971), 221-27.

[6] N. Wirth. Systematic Programming. Englewood Cliffs, N.J. Prentice-Hall, Inc., 1973.

[7] K. Jensen and N. Wirth. PASCAL-User Manual and Report. Berlin, Heidelberg, New York; Springer-Verlag, 1974.

Preface To The 1985 Edition

This new Edition incorporates many revisions of details and several changes of more significant nature. They were all motivated by experiences made in the ten years since the first Edition appeared. Most of the contents and the style of the text, however, have been retained. We briefly summarize the major alterations. The major change which pervades the entire text concerns the programming language used to express the algorithms. Pascal has been replaced by Modula-2. Although this change is of no fundamental influence to the presentation of the algorithms, the choice is justified by the simpler and more elegant syntactic structures of Modula-2, which often lead to a more lucid representation of an algorithm's structure. Apart from this, it appeared advisable to use a notation that is rapidly gaining acceptance by a wide community, because it is well-suited for the development of large programming systems. Nevertheless, the fact that Pascal is Modula's ancestor is very evident and eases the task of a transition. The syntax of Modula is summarized in the Appendix for easy reference.

As a direct consequence of this change of programming language, Sect. 1.11 on the sequential file structure has been rewritten. Modula-2 does not offer a built-in file type. The revised Sect. 1.11 presents the concept of a sequence as a data structure in a more general manner, and it introduces a set of program modules that incorporate the sequence concept in Modula-2 specifically.

The last part of Chapter 1 is new. It is dedicated to the subject of searching and, starting out with linear and binary search, leads to some recently invented fast string searching algorithms. In this section in particular we use assertions and loop invariants to demonstrate the correctness of the presented algorithms.

A new section on priority search trees rounds off the chapter on dynamic data structures. Also this species of trees was unknown when the first Edition appeared. They allow an economical representation and a fast search of point sets in a plane.

The entire fifth chapter of the first Edition has been omitted. It was felt that the subject of compiler construction was somewhat isolated from the preceding chapters and would rather merit a more extensive treatment in its own volume.

Finally, the appearance of the new Edition reflects a development that has profoundly influenced publications in the last ten years: the use of computers and sophisticated algorithms to prepare and automatically typeset documents. This book was edited and laid out by the author with the aid of a Lilith computer and its document editor Lara. Without these tools, not only would the book become more costly, but it would certainly not be finished yet.

Palo Alto, March 1985 N. Wirth

Notation

The following notations, adopted from publications of E.W. Dijkstra, are used in this book.

In logical expressions, the character & denotes conjunction and is pronounced as and. The character ~ denotes negation and is pronounced as not. Boldface **A** and **E** are used to denote the universal and existential quantifiers. In the following formulas, the left part is the notation used and defined here in terms of the right part. Note that the left parts avoid the use of the symbol "...", which appeals to the readers intuition.

$\mathbf{A}i: m \le i < n : P_i \qquad P_m \,\&\, P_{m+1} \,\&\, \ldots \,\&\, P_{n-1}$

The P_i are predicates, and the formula asserts that for all indices i ranging from a given value m to, but excluding a value n P_i holds.

$\mathbf{E}i: m \le i < n : P_i \qquad P_m \text{ or } P_{m+1} \text{ or } \ldots \text{ or } P_{n-1}$

The P_i are predicates, and the formula asserts that for some indices i ranging from a given value m to, but excluding a value n P_i holds.

$\mathbf{S}i: m \le i < n : x_i = x_m + x_{m+1} + \ldots + x_{n-1}$

$\text{MIN } i: m \le i < n : x_i = \text{minimum}(x_m , \ldots , x_{n-1})$

$\text{MAX } i: m \le i < n : x_i = \text{maximum}(x_m , \ldots , x_{n-1})$

1 Fundamental Data Structures

1.1 Introduction

The modern digital computer was invented and intended as a device that should facilitate and speed up complicated and time-consuming computations. In the majority of applications its capability to store and access large amounts of information plays the dominant part and is considered to be its primary characteristic, and its ability to compute, i.e., to calculate, to perform arithmetic, has in many cases become almost irrelevant.

In all these cases, the large amount of information that is to be processed in some sense represents an abstraction of a part of reality. The information that is available to the computer consists of a selected set of data about the actual problem, namely that set that is considered relevant to the problem at hand, that set from which it is believed that the desired results can be derived. The data represent an abstraction of reality in the sense that certain properties and characteristics of the real objects are ignored because they are peripheral and irrelevant to the particular problem. An abstraction is thereby also a simplification of facts.

We may regard a personnel file of an employer as an example. Every employee is represented (abstracted) on this file by a set of data relevant either to the employer or to his accounting procedures. This set may include some identification of the employee, for example, his or her name and salary. But it will most probably not include irrelevant data such as the hair color, weight, and height.

In solving a problem with or without a computer it is necessary to choose an abstraction of reality, i.e., to define a set of data that is to represent the real situation. This choice must be guided by the problem to be solved. Then follows a choice of representation of this information. This choice is guided by the tool that is to solve the problem, i.e., by the facilities offered by the computer. In most cases these two steps are not entirely separable.

The choice of representation of data is often a fairly difficult one, and it is not uniquely determined by the facilities available. It must always be taken in the light of the operations that are to be performed on the data. A good example is the representation of numbers, which are themselves abstractions of properties of objects to be characterized. If addition is the only (or at least the dominant) operation to be performed, then a good way to represent the number n is to write n strokes. The addition rule on this representation is indeed very obvious and simple. The Roman numerals are based on the same principle of simplicity, and the adding rules are similarly straightforward for small numbers. On the other hand, the representation by Arabic numerals requires rules that are far from obvious (for small numbers) and they must be memorized. However, the situation is reversed when we consider either addition of large numbers or multiplication and division. The decomposition of these operations into simpler ones is much easier in the case of representation by Arabic numerals because of their systematic structuring principle that is based on positional weight of the digits.

It is generally known that computers use an internal representation based on binary digits (bits). This representation is unsuitable for human beings because of the usually large number of digits involved, but it is most suitable for electronic circuits because the two values 0 and 1 can be represented conveniently and reliably by the presence or absence of electric currents, electric charge, or magnetic fields.

From this example we can also see that the question of representation often transcends several levels of detail. Given the problem of representing, say, the position of an object, the first decision may lead to the choice of a pair of real numbers in, say, either Cartesian or polar coordinates. The second decision may lead to a floating-point representation, where every real number x consists of a pair of integers denoting a fraction f and an exponent e to a certain base (such that $x = f \times 2^e$). The third decision, based on the

knowledge that the data are to be stored in a computer, may lead to a binary, positional representation of integers, and the final decision could be to represent binary digits by the electric charge in a semiconductor storage device. Evidently, the first decision in this chain is mainly influenced by the problem situation, and the later ones are progressively dependent on the tool and its technology. Thus, it can hardly be required that a programmer decide on the number representation to be employed, or even on the storage device characteristics. These lower-level decisions can be left to the designers of computer equipment, who have the most information available on current technology with which to make a sensible choice that will be acceptable for all (or almost all) applications where numbers play a role.

In this context, the significance of programming languages becomes apparent. A programming language represents an abstract computer capable of interpreting the terms used in this language, which may embody a certain level of abstraction from the objects used by the actual machine. Thus, the programmer who uses such a higher-level language will be freed (and barred) from questions of number representation, if the number is an elementary object in the realm of this language.

The importance of using a language that offers a convenient set of basic abstractions common to most problems of data processing lies mainly in the area of reliability of the resulting programs. It is easier to design a program based on reasoning with familiar notions of numbers, sets, sequences, and repetitions than on bits, storage units, and jumps. Of course, an actual computer represents all data, whether numbers, sets, or sequences, as a large mass of bits. But this is irrelevant to the programmer as long as he or she does not have to worry about the details of representation of the chosen abstractions, and as long as he or she can rest assured that the corresponding representation chosen by the computer (or compiler) is reasonable for the stated purposes.

The closer the abstractions are to a given computer, the easier it is to make a representation choice for the engineer or implementor of the language, and the higher is the probability that a single choice will be suitable for all (or almost all) conceivable applications. This fact sets definite limits on the degree of abstraction from a given real computer. For example, it would not make sense to include geometric objects as basic data items in a general-purpose language, since their proper repesentation will, because of its inherent complexity, be largely dependent on the operations to be applied to these objects. The nature and frequency of these operations will, however, not be known to the designer of a general-purpose language and its compiler, and any choice the designer makes may be inappropriate for some potential applications.

In this book these deliberations determine the choice of notation for the description of algorithms and their data. Clearly, we wish to use familiar notions of mathematics, such as numbers, sets, sequences, and so on, rather than computer-dependent entities such as bitstrings. But equally clearly we wish to use a notation for which efficient compilers are known to exist. It is equally unwise to use a closely machine-oriented and machine-dependent language, as it is unhelpful to describe computer programs in an abstract notation that leaves problems of representation widely open. The programming language Pascal had been designed in an attempt to find a compromise between these extremes, and the successor languages Modula-2 and Oberon are the result of decades of experience [1-3]. Oberon retains Pascal's basic concepts and incorporates some improvements and some extensions; it is used throughout this book [1-5]. It has been successfully implemented on several computers, and it has been shown that the notation is sufficiently close to real machines that the chosen features and their representations can be clearly explained. The language is also sufficiently close to other languages, and hence the lessons taught here may equally well be applied in their use.

1.2 The Concept of Data Type

In mathematics it is customary to classify variables according to certain important characteristics. Clear distinctions are made between real, complex, and logical variables or between variables representing individual values, or sets of values, or sets of sets, or between functions, functionals, sets of functions, and

so on. This notion of classification is equally if not more important in data processing. We will adhere to the principle that every constant, variable, expression, or function is of a certain *type*. This type essentially characterizes the set of values to which a constant belongs, or which can be assumed by a variable or expression, or which can be generated by a function.

In mathematical texts the type of a variable is usually deducible from the typeface without consideration of context; this is not feasible in computer programs. Usually there is one typeface available on computer equipment (i.e., Latin letters). The rule is therefore widely accepted that the associated type is made explicit in a declaration of the constant, variable, or function, and that this *declaration* textually precedes the application of that constant, variable, or function. This rule is particularly sensible if one considers the fact that a compiler has to make a choice of representation of the object within the store of a computer. Evidently, the amount of storage allocated to a variable will have to be chosen according to the size of the range of values that the variable may assume. If this information is known to a compiler, so-called dynamic storage allocation can be avoided. This is very often the key to an efficient realization of an algorithm.

The primary characteristics of the concept of type that is used throughout this text, and that is embodied in the programming language Oberon, are the following [1-2]:

1. A data type determines the set of values to which a constant belongs, or which may be assumed by a variable or an expression, or which may be generated by an operator or a function.
2. The type of a value denoted by a constant, variable, or expression may be derived from its form or its declaration without the necessity of executing the computational process.
3. Each operator or function expects arguments of a fixed type and yields a result of a fixed type. If an operator admits arguments of several types (e.g., + is used for addition of both integers and real numbers), then the type of the result can be determined from specific language rules.

As a consequence, a compiler may use this information on types to check the legality of various constructs. For example, the mistaken assignment of a Boolean (logical) value to an arithmetic variable may be detected without executing the program. This kind of redundancy in the program text is extremely useful as an aid in the development of programs, and it must be considered as the primary advantage of good high-level languages over machine code (or symbolic assembly code). Evidently, the data will ultimately be represented by a large number of binary digits, irrespective of whether or not the program had initially been conceived in a high-level language using the concept of type or in a typeless assembly code. To the computer, the store is a homogeneous mass of bits without apparent structure. But it is exactly this abstract structure which alone is enabling human programmers to recognize meaning in the monotonous landscape of a computer store.

The theory presented in this book and the programming language Oberon specify certain methods of defining data types. In most cases new data types are defined in terms of previously defined data types. Values of such a type are usually conglomerates of component values of the previously defined constituent types, and they are said to be *structured*. If there is only one constituent type, that is, if all components are of the same constituent type, then it is known as the base type. The number of distinct values belonging to a type `T` is called its *cardinality*. The cardinality provides a measure for the amount of storage needed to represent a variable `x` of the type `T`, denoted by `x: T`.

Since constituent types may again be structured, entire hierarchies of structures may be built up, but, obviously, the ultimate components of a structure are atomic. Therefore, it is necessary that a notation is provided to introduce such primitive, unstructured types as well. A straightforward method is that of *enumerating* the values that are to constitute the type. For example in a program concerned with plane geometric figures, we may introduce a primitive type called shape, whose values may be denoted by the identifiers *rectangle, square, ellipse, circle*. But apart from such programmer-defined types, there will

have to be some standard, predefined types. They usually include numbers and logical values. If an ordering exists among the individual values, then the type is said to be ordered or scalar. In Oberon, all unstructured types are ordered; in the case of explicit enumeration, the values are assumed to be ordered by their enumeration sequence.

With this tool in hand, it is possible to define primitive types and to build conglomerates, structured types up to an arbitrary degree of nesting. In practice, it is not sufficient to have only one general method of combining constituent types into a structure. With due regard to practical problems of representation and use, a general-purpose programming language must offer several methods of structuring. In a mathematical sense, they are equivalent; they differ in the operators available to select components of these structures. The basic structuring methods presented here are the *array*, the *record*, the *set*, and the *sequence*. More complicated structures are not usually defined as static types, but are instead dynamically generated during the execution of the program, when they may vary in size and shape. Such structures are the subject of Chap. 4 and include lists, rings, trees, and general, finite graphs.

Variables and data types are introduced in a program in order to be used for computation. To this end, a set of operators must be available. For each standard data type a programming languages offers a certain set of primitive, standard operators, and likewise with each structuring method a distinct operation and notation for selecting a component. The task of composition of operations is often considered the heart of the art of programming. However, it will become evident that the appropriate composition of data is equally fundamental and essential.

The most important basic operators are comparison and assignment, i.e., the test for equality (and for order in the case of ordered types), and the command to enforce equality. The fundamental difference between these two operations is emphasized by the clear distinction in their denotation throughout this text.

Test for equality: `x = y` (an expression with value TRUE or FALSE)

Assignment to x: `x := y` (a statement making x equal to y)

These fundamental operators are defined for most data types, but it should be noted that their execution may involve a substantial amount of computational effort, if the data are large and highly structured.

For the standard primitive data types, we postulate not only the availability of assignment and comparison, but also a set of operators to create (compute) new values. Thus we introduce the standard operations of arithmetic for numeric types and the elementary operators of propositional logic for logical values.

1.3 Standard Primitive Types

Standard primitive types are those types that are available on most computers as built-in features. They include the whole numbers, the logical truth values, and a set of printable characters. On many computers fractional numbers are also incorporated, together with the standard arithmetic operations. We denote these types by the identifiers

INTEGER, REAL, BOOLEAN, CHAR, SET

1.3.1 The type INTEGER

The type INTEGER comprises a subset of the whole numbers whose size may vary among individual computer systems. If a computer uses n bits to represent an integer in two's complement notation, then the admissible values x must satisfy $-2^{n-1} \leq x < 2^{n-1}$. It is assumed that all operations on data of this type are exact and correspond to the ordinary laws of arithmetic, and that the computation will be interrupted in the case of a result lying outside the representable subset. This event is called *overflow*. The standard operators are the four basic arithmetic operations of addition (+), subtraction (-), multiplication (*) and

division (/, DIV).

Whereas the slash denotes ordinary division resulting in a value of type REAL, the operator DIV denotes integer division resulting in a value of type INTEGER. If we define the quotient q = m DIV n and the remainder r = m MOD n, the following relations hold, assuming n > 0:

$q*n + r = m$ and $0 \le r < n$

Examples

```
 31 DIV 10  =  3      31 MOD 10  =  1
-31 DIV 10  = -4     -31 MOD 10  =  9
```

We know that dividing by 10^n can be achieved by merely shifting the decimal digits n places to the right and thereby ignoring the lost digits. The same method applies, if numbers are represented in binary instead of decimal form. If two's complement representation is used (as in practically all modern computers), then the shifts implement a division as defined by the above DIV operaton. Moderately sophisticated compilers will therefore represent an operation of the form m DIV 2^n or m MOD 2^n by a fast shift (or mask) operation.

1.3.2 The type REAL

The type REAL denotes a subset of the real numbers. Whereas arithmetic with operands of the types INTEGER is assumed to yield exact results, arithmetic on values of type REAL is permitted to be inaccurate within the limits of round-off errors caused by computation on a finite number of digits. This is the principal reason for the explicit distinction between the types INTEGER and REAL, as it is made in most programming languages.

The standard operators are the four basic arithmetic operations of addition (+), subtraction (-), multiplication (*), and division (/). It is an essence of data typing that different types are incompatible under assignment. An exception to this rule is made for assignment of integer values to real variables, because here the semanitcs are unambiguous. After all, integers form a subset of real numbers. However, the inverse direction is not permissible: Assignment of a real value to an integer variable requires an operation such as truncation or rounding. The standard transfer function ENTIER(x) yields the integral part of x. Rounding of x is obtained by ENTIER(x + 0.5).

Many programming languages do not include an exponentiation operator. The following is an algorithm for the fast computation of $y = x^n$, where n is a non-negative integer.

```
y := 1.0; i := n;                                        (* ADenS13 *)
WHILE i > 0 DO   (* x0^n = x^i * y *)
  IF ODD(i) THEN y := y*x END;
  x := x*x; i := i DIV 2
END
```

1.3.3 The type BOOLEAN

The two values of the standard type BOOLEAN are denoted by the identifiers TRUE and FALSE. The Boolean operators are the logical conjunction, disjunction, and negation whose values are defined in Table 1.1. The logical conjunction is denoted by the symbol &, the logical disjunction by OR, and negation by "~". Note that comparisons are operations yielding a result of type BOOLEAN. Thus, the result of a comparison may be assigned to a variable, or it may be used as an operand of a logical operator in a Boolean expression. For instance, given Boolean variables p and q and integer variables x = 5, y = 8, z = 10, the two assignments

```
p := x = y
q := (x ≤ y) & (y < z)
```

yield p = FALSE and q = TRUE.

p	q	p OR q	p & q	~p
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	TRUE	FALSE	FALSE
FALSE	TRUE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

Table 1.1. Boolean Operators.

The Boolean operators & (AND) and OR have an additional property in most programming languages, which distinguishes them from other dyadic operators. Whereas, for example, the sum x+y is not defined, if either x or y is undefined, the conjunction p&q is defined even if q is undefined, provided that p is FALSE. This conditionality is an important and useful property. The exact definition of & and OR is therefore given by the following equations:

```
p & q    =  if p then q else FALSE
p OR q   =  if p then TRUE else q
```

1.3.4 The type CHAR

The standard type CHAR comprises a set of printable characters. Unfortunately, there is no generally accepted standard character set used on all computer systems. Therefore, the use of the predicate "standard" may in this case be almost misleading; it is to be understood in the sense of "standard on the computer system on which a certain program is to be executed."

The character set defined by the International Standards Organization (ISO), and particularly its American version ASCII (American Standard Code for Information Interchange) is the most widely accepted set. The ASCII set is therefore tabulated in Appendix A. It consists of 95 printable (graphic) characters and 33 control characters, the latter mainly being used in data transmission and for the control of printing equipment.

In order to be able to design algorithms involving characters (i.e., values of type CHAR), that are system independent, we should like to be able to assume certain minimal properties of character sets, namely:

1. The type CHAR contains the 26 capital Latin letters, the 26 lower-case letters, the 10 decimal digits, and a number of other graphic characters, such as punctuation marks.

2. The subsets of letters and digits are ordered and contiguous, i.e.,

("A" ≤ x) & (x ≤ "Z") implies that x is a capital letter

("a" ≤ x) & (x ≤ "z") implies that x is a lower-case letter

("0" ≤ x) & (x ≤ "9") implies that x is a decimal digit

3. The type CHAR contains a non-printing, blank character and a line-end character that may be used as separators.

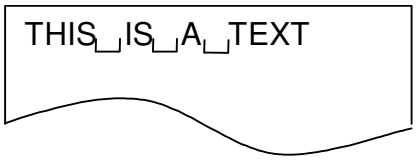

Fig. 1.1. Representations of a text

The availability of two standard type transfer functions between the types CHAR and INTEGER is particularly important in the quest to write programs in a machine independent form. We will call them ORD(ch), denoting the ordinal number of ch in the character set, and CHR(i), denoting the character with ordinal number i. Thus, CHR is the inverse function of ORD, and vice versa, that is,

ORD(CHR(i)) = i (if CHR(i) is defined)

CHR(ORD(c)) = c

Furthermore, we postulate a standard function CAP(ch). Its value is defined as the capital letter corresponding to ch, provided ch is a letter.

ch is a lower-case letter implies that CAP(ch) = corresponding capital letter

ch is a capital letter implies that CAP(ch) = ch

1.3.5 The type SET

The type SET denotes sets whose elements are integers in the range 0 to a small number, typically 31 or 63. Given, for example, variables

```
VAR r, s, t: SET
```

possible assignments are

```
r := {5}; s := {x, y .. z}; t := {}
```

Here, the value assigned to r is the singleton set consisting of the single element 5; to t is assigned the empty set, and to s the elements x, y, y+1, … , z-1, z.

The following elementary operators are defined on variables of type SET:

*	set intersection
+	set union
-	set difference
/	symmetric set difference
IN	set membership

Constructing the intersection or the union of two sets is often called set multiplication or set addition, respectively; the priorities of the set operators are defined accordingly, with the intersection operator having priority over the union and difference operators, which in turn have priority over the membership

operator, which is classified as a relational operator. Following are examples of set expressions and their fully parenthesized equivalents:

```
r * s + t   = (r*s) + t
r - s * t   = r - (s*t)
r - s + t   = (r-s) + t
r + s / t   = r + (s/t)
x IN s + t  = x IN (s+t)
```

1.4 The Array Structure

The array is probably the most widely used data structure; in some languages it is even the only one available. An array consists of components which are all of the same type, called its base type; it is therefore called a *homogeneous* structure. The array is a *random-access* structure, because all components can be selected at random and are equally quickly accessible. In order to denote an individual component, the name of the entire structure is augmented by the *index* selecting the component. This index is to be an integer between 0 and n-1, where n is the number of elements, the *size*, of the array.

```
TYPE T = ARRAY n OF T0
```

Examples

```
TYPE Row    = ARRAY 4 OF REAL
TYPE Card   = ARRAY 80 OF CHAR
TYPE Name   = ARRAY 32 OF CHAR
```

A particular value of a variable

```
VAR x: Row
```

with all components satisfying the equation $x_i = 2^{-i}$, may be visualized as shown in Fig. 1.2.

x_0	1.0
x_1	0.5
x_2	0.25
x_3	0.125

Fig. 1.2. Array of type Row with $x_i = 2^{-i}$.

An individual component of an array can be selected by an *index*. Given an array variable x, we denote an array selector by the array name followed by the respective component's index i, and we write x_i or x[i]. Because of the first, conventional notation, a component of an array component is therefore also called a *subscripted* variable.

The common way of operating with arrays, particularly with large arrays, is to selectively update single components rather than to construct entirely new structured values. This is expressed by considering an array variable as an array of component variables and by permitting assignments to selected components, such as for example x[i] := 0.125. Although selective updating causes only a single component value to change, from a conceptual point of view we must regard the entire composite value as having changed too.

The fact that array indices, i.e., names of array components, are integers, has a most important consequence: indices may be computed. A general index expression may be substituted in place of an index constant; this expression is to be evaluated, and the result identifies the selected component. This

generality not only provides a most significant and powerful programming facility, but at the same time it also gives rise to one of the most frequently encountered programming mistakes: The resulting value may be outside the interval specified as the range of indices of the array. We will assume that decent computing systems provide a warning in the case of such a mistaken access to a non-existent array component.

The cardinality of a structured type, i. e. the number of values belonging to this type, is the product of the cardinality of its components. Since all components of an array type T are of the same base type T0, we obtain

$card(T) = card(T0)^n$

Constituents of array types may themselves be structured. An array variable whose components are again arrays is called a *matrix*. For example,

```
M: ARRAY 10 OF Row
```

is an array consisting of ten components (rows), each constisting of four components of type REAL. and is called a 10 x 4 matrix with real components. Selectors may be concatenated accordingly, such that M_{ij} and M[i][j] denote the j-th component of row M_i, which is the i-th component of M. This is usually abbreviated as M[i,j], and in the same spirit the declaration

```
M: ARRAY 10 OF ARRAY 4 OF REAL
```

can be written more concisely as

```
M: ARRAY 10, 4 OF REAL
```

If a certain operation has to be performed on all components of an array or on adjacent components of a section of the array, then this fact may conveniently be emphasized by using the FOR satement, as shown in the following examples for computing the sum and for finding the maximal element of an array declared as

```
VAR a: ARRAY N OF INTEGER;                                    (* ADenS14 *)

sum := 0;
FOR i := 0 TO N-1 DO sum := a[i] + sum END

k := 0; max := a[0];
FOR i := 1 TO N-1 DO
   IF max < a[i] THEN k := i; max := a[k] END
END
```

In a further example, assume that a fraction f is represented in its decimal form with k-1 digits, i.e., by an array d such that

$f = \mathbf{S}i: 0 \le i < k: d_i * 10^{-i}$

$f = d_0 + 10*d_1 + 100*d_2 + \ldots + 10^{k-1}*d_{k-1}$

Now assume that we wish to divide f by 2. This is done by repeating the familiar division operation for all k-1 digits d_i, starting with i=1. It consists of dividing each digit by 2 taking into account a possible carry from the previous position, and of retaining a possible remainder r for the next position:

```
r := 10*r +d[i];  d[i] := r DIV 2;  r := r MOD 2
```

This algorithm is used to compute a table of negative powers of 2. The repetition of halving to compute 2^{-1}, 2^{-2}, ... , 2^{-N} is again appropriately expressed by a FOR statement, thus leading to a nesting of two FOR statements.

```
PROCEDURE Power (VAR W: Texts.Writer; N: INTEGER);          (* ADenS14 *)
  (*compute decimal representation of negative powers of 2*)
  VAR i, k, r: INTEGER;
    d: ARRAY N OF INTEGER;
BEGIN
  FOR k := 0 TO N-1 DO
    Texts.Write(W, "."); r := 0;
    FOR i := 0 TO k-1 DO
      r := 10*r + d[i]; d[i] := r DIV 2; r := r MOD 2;
      Texts.Write(W, CHR(d[i] + ORD("0")))
    END;
    d[k] := 5; Texts.Write(W, "5"); Texts.WriteLn(W)
  END
END Power
```

The resulting output text for N = 10 is

```
.5
.25
.125
.0625
.03125
.015625
.0078125
.00390625
.001953125
.0009765625
```

1.5 The Record Structure

The most general method to obtain structured types is to join elements of arbitrary types, that are possibly themselves structured types, into a compound. Examples from mathematics are complex numbers, composed of two real numbers, and coordinates of points, composed of two or more numbers according to the dimensionality of the space spanned by the coordinate system. An example from data processing is describing people by a few relevant characteristics, such as their first and last names, their date of birth, sex, and marital status.

In mathematics such a compound type is the Cartesian product of its constituent types. This stems from the fact that the set of values defined by this compound type consists of all possible combinations of values, taken one from each set defined by each constituent type. Thus, the number of such combinations, also called *n-tuples*, is the product of the number of elements in each constituent set, that is, the cardinality of the compound type is the product of the cardinalities of the constituent types.

In data processing, composite types, such as descriptions of persons or objects, usually occur in files or data banks and record the relevant characteristics of a person or object. The word record has therefore become widely accepted to describe a compound of data of this nature, and we adopt this nomenclature in preference to the term Cartesian product. In general, a record type T with components of the types T_1, T_2, ... , T_n is defined as follows:

```
TYPE T   = RECORD  s1: T1; s2: T2;  ...  sn: Tn  END
card(T)  = card(T1) * card(T2) * ... * card(Tn)
```

Examples

```
TYPE Complex =  RECORD  re, im: REAL END
TYPE Date  =    RECORD  day, month, year:  INTEGER END
TYPE Person  =  RECORD  name, firstname: Name;
                        birthdate: Date;
                        male: BOOLEAN
                END
```

We may visualize particular, record-structured values of, for example, the variables

```
z: Complex
d: Date
p: Person
```

as shown in Fig. 1.3.

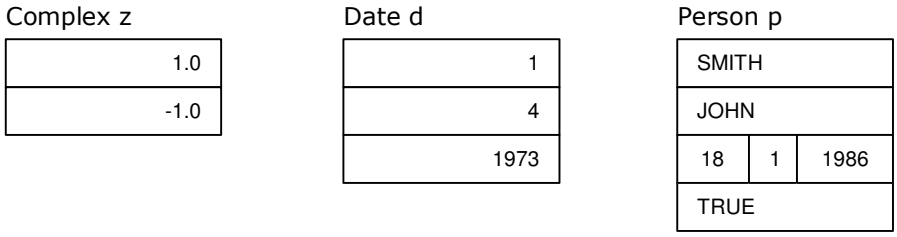

Fig. 1.3. Records of type Complex, Date and Person.

The identifiers s_1, s_2, ... , s_n introduced by a record type definition are the names given to the individual components of variables of that type. As components of records are called *fields*, the names are *field identifiers*. They are used in record selectors applied to record structured variables. Given a variable x: T, its i-th field is denoted by $x.s_i$. Selective updating of x is achieved by using the same selector denotation on the left side in an assignment statement:

$x.s_i$:= e

where e is a value (expression) of type T_i. Given, for example, the record variables z, d, and p declared above, the following are selectors of components:

```
z.im              (of type REAL)
d.month           (of type INTEGER)
p.name            (of type Name)
p.birthdate       (of type Date)
p.birthdate.day   (of type INTEGER)
p.mail            (of type BOOLEAN)
```

The example of the type Person shows that a constituent of a record type may itself be structured. Thus, selectors may be concatenated. Naturally, different structuring types may also be used in a nested fashion. For example, the i-th component of an array a being a component of a record variable r is denoted by r.a[i], and the component with the selector name s of the i-th record structured component of the array a is denoted by a[i].s.

It is a characteristic of the Cartesian product that it contains all combinations of elements of the constituent types. But it must be noted that in practical applications not all of them may be meaningful. For instance, the type Date as defined above includes the 31st April as well as the 29th February 1985, which are both dates that never occurred. Thus, the definition of this type does not mirror the actual situation