Medizinische Informatik und Statistik

Herausgeber: S. Koller, P. L. Reichertz und K. Überla

3

Informatics and Medicine

An Advanced Course

Edited by
P. L. Reichertz and G. Goos

Springer-Verlag
Berlin · Heidelberg · New York 1977

Bandherausgeber
Prof. Dr. P. L. Reichertz
Abteilung und Lehrstuhl für
Medizinische Informatik
Medizinische Hochschule Hannover
Karl-Wiechert-Allee 9
3000 Hannover 61

Prof. Dr. G. Goos
Universität Karlsruhe
Institut für Informatik II
Postfach 6380
7500 Karlsruhe 1

Library of Congress Cataloging in Publication Data
Main entry under title:

Informatics and medicine.

(Medizinische Informatik und Statistik ; 3)
1. Medicine--Data processing--Congresses.
2. Medicine--Documentation--Congresses. I. Reichertz,
P.L, 1930- II. Goos, Gerhard, 1937-
III. Series.
R858.A1I46 362.1'028'54 77-660

ISBN-13:978-3-540-08120-3 e-ISBN-13:978-3-642-81110-4
DOI: 10.1007/978-3-642-81110-4

Table of Contents

Preface

The modern development of medicine has been characterized by the growing use of new technologies in health care delivery and research. As an empirical science, medicine is based on many types and quantities of information to recognize alterations, explore causes and apply corrective action. Dealing with biological objects, signals have to be collected, processed and interpreted to recognize the state of this object. It is therefore understandable that data processing technology and informatics have been employed to a growing extent in medicine. The increasing economic repercussions of modern medicine lead also to the demand of ways and means to assess the system as such and to develop means for evaluation and regulation.

However, the application of data processing to the medical field has very often grown in parallel to and remote from the development of informatics and data processing in general. Furthermore, difficulties have occurred resulting from the differing concepts of reasoning, decision making and methodology. We therefore decided to start a series of seminars with the attempt to bring scientists from both medicine and informatics together to discuss basic principles of informatics and medicine and to attempt a synthesis between the problems in medicine and health care delivery and methods in informatics to approach a solution of these problems. This volume contains the lecture notes of the first seminar of this type. The goals of this advanced course were:

- to convey knowledge about general and specific problems in theoretical medicine and health care delivery which are suitable for or subject to the application of methods and technology of informatics and data processing,

- to identify and teach systematized concepts and methodology in informatics which relate to the problems and the application of which may open more efficient ways for the solution of these problems in particular and broaden and solidify the knowledge of those who work in leading positions in medical informatics in general.

It is unnecessary to mention that neither the broad field of medicine nor all developments in informatics could be treated during this seminar. Therefore, this volume is a selection of topics presented in a way which still show the inhomogeneity of approaches. The editors hope that this publication will serve as a basis for further seminars of the same type and as a condensation point furthering the development of an adequate methodology to solve specific and general problems in theoretical medicine and health care delivery with its many ramifications.

We are greatly indebted to Mrs. U. Piccolo for her help in preparing this publication and to Drs. Engelbrecht and Wolters who organized the course.

P.L. Reichertz/ G. Goos

Contributors:

ALBER, K., Prof. Dr.
Lehrstuhl A für Informatik
Universität Braunschweig
Gaußstr. 12
D-3300 Braunschweig
Fed. Rep. Germany

ANDERSON, J., M.D.
Professor of Medicine
King's College Hospital Medical School
Denmark Hill
London SE5 8RX
Great Britain

GOLDBERG, M., Dr.
Département de Biophysique
et de Biomathématiques
Faculté de Médecine Pitie-Salpetrière
Boulevard de l'Hôpital
F-75634 Paris Cedex 13
France

GOOS, G., Prof. Dr.
Institut für Informatik
Universität Karlsruhe
Zirkel 2
D-7500 Karlsruhe
Fed. Rep. Germany

GREMY, F. Prof. Dr.
Département de Biophysique
et de Biomathématiques
Faculté de Medecine Pitie-Salpetrière
Boulevard de l'Hôpital
F-75634 Paris Cedex 13
France

GRIFFITHS, M., Prof. Dr.
Departement d'Informatique
I.U.T. de Nancy
2 bis Boulevard Charlemagne
F-5400 Nancy
France

HARTMANN, F., Prof. Dr.
Department Innere Medizin
Medizinische Hochschule Hannover
Karl-Wiechert-Allee 9
D-3000 Hannover 61
Fed. Rep. Germany

LEILICH, H.-O., Prof. Dr.
Institut für Mathematik
Universität Braunschweig
Gaußstr. 28
D-3300 Braunschweig
Fed. Rep. Germany

LOCKEMANN, P.C., Prof. Dr.
Institut für Informatik
Universität Karlsruhe
Zirkel 2
D-7500 Karlsruhe

LODWICK, G.S., M.D.
Professor and Chairman
Department of Radiology
University of Missouri Medical Center
Columbia, Mo 65201
USA

MÖHR, J.R., Priv. Doz. Dr.
Institut für Med. Informatik
Medizinische Hochschule Hannover
Karl-Wiechert-Allee 9
D-3000 Hannover 61
Fed. Rep. Germany

REICHERTZ, P.L., Prof. Dr.
Institut für Med. Informatik
Medizinische Hochschule Hannover
Karl-Wiechert-Allee 9
D-3000 Hannover 61
Fed. Rep. Germany

SCHNEIDER, W., Prof. Dr.
Uppsala Datacentral
Box 2103
S-75002 Uppsala
Sweden

VALLBONA, C., M.D.
Professor and Chairman
Department of Community Medicine
Baylor College of Medicine
Texas Medical Center
Houston, Texas 77025
U.S.A.

WINGERT, F., Prof. Dr.
Institut für Med. Informatik
und Biomathematik
Hüfferstr. 75
D-4400 Münster
Fed. Rep. Germany

BASIC NOTIONS OF INFORMATICS

Gerhard Goos, Karlsruhe

0. Introduction

Informatics is concerned with information processing as supported by technical tools. Today's major technical tool for processing information is the computer. Hence, to a large extent informatics is concerned with computers, their construction and their applications. In the latter area informatics is mainly interested in extracting the common underlying principles of different fields of application and in generalizing those principles to apply them to other fields. This interest is justified by our experience that many applications which, at first sight, seem to not have anything in common, rely on the same methods and can take over many principles from each other.

Historically, informatics arose as an independent science when the growing significance of computers required a deeper understanding of technical information processing. Many insights, however, which were gained in the last decade have a much wider range of applicability than that indicated by computers.

The first part of this lecture discusses the notion of information and its technical representation as data, the notion of algorithm as the fundamental means for describing information processing, and the relationship between algorithms and data. The second part is concerned with how to present algorithms and data to computers and with the development of algorithms for computers.

1. Information and its technical representation by messages

The English word "morning" and the German word "Morgen" represent to those who understand these languages the same information. Without knowledge of these languages this information cannot be derived. We learn from such examples that information is an abstract entity which may be given in various ways by concrete encodings which we call messages. The relationship between a message and its information cannot be deduced; it is a convention which must be learned.

The example further shows that there may be many representations of the same information by distinct messages. Conversely there may be distinct informations which are represented by the same message. Which information is deduced may depend on the context or on the people, or the message is just ambiguous. Whether the sign $\times$ denotes the letter x or a multiplication sign in a mathematical formula depends on the rest

of the formula. The letter ε has a very distinguished meaning in mathematics which is usually not deduced by a non-mathematician; the broadcast announcement of a traffic jam carries completely different information for those who are on the road in question than for others. Examples of ambiguities are such well-known sentences as

they are flying planes

or

fruit flies like a banana.

To give a more technical example: 11110000 may be the binary encoding of the digit zero in EBCDI-Code or it may be the binary encoding of the number 240. Also we note that a given message-information relation may hold with a certain probability. Examples are the informations about illnesses deduced from certain chemical determinations in the clinical laboratory.

In conclusion we see that the relationship between messages and informations is a many-to-many relation. An actual relationship is selected by giving an interpretation rule α which states the information $I = \alpha(N)$ for each message N of a class of messages:

$$\alpha : \mathbb{N} \rightarrow \mathbb{I}.$$

The interpretation rule is usually given informally, e.g., if someoody understands medical terminology then this means that he knows the interpretation rule for a set of specific medical terms. The interpretation rule could also be stated formally or even by the hardware or software of a computer. For example, a computer in doing floating-point arithmetic shows that it knows how to interpret a bit pattern as a floating-point number.

As the outcome of this discussion, however, we must realize that information processing with technical tools always means message processing with technical tools. If we send informations to a system for automatic information processing e.g., to a computer, then in fact we present messages to the computer which we believe can be interpreted as the required information. If we take results from a computer then we have to interpret them to deriving the information which we require. Also by reasoning about the possible operations which a computer can perform, we easily see that it can only handle messages and can never handle information on the user level.

This insight is important because it shows that in addition to the automatic processing device, we need interpretation rules for how to encode information as input to this device and how to decode the informations from the result messages. A computer never processes medical data but at best bit patterns which might be interpreted as

medical data. All too often the computer is blamed for mistakes which result from a lack of knowledge and imprecision in applying the interpretation rules for input or output data.

The interpretation of messages may consist of several hierarchically ordered steps. For example, if we interpret 11 as the binary equivalent of the number 3 then in turn we might see that this is the number of unknowns in a set of mathematical equations, and this value might be implied by the number of dimensions in physical space. In fact, a computer does not work with bits or numbers or strings of characters, for all of these are informations which occur by interpreting certain physical processes, impulses, magnetic states etc. in a suitable manner. We see therefore that whatever we consider as the underlying message M of an information, appears, from another point of view, to be itself an information. Hence, when we say, M is a message, we characterize a certain level of abstraction by showing that it is presently of no concern to us that M itself might be the interpretation of a more elementary message.

2. The Notion of Algorithm

We know now that every information which we wish to process automatically must be encoded as a message and that the result is another message. The word data is commonly used for messages which are the subject or result of such processes. Our next question is: "What are the characteristic properties of processes which can be done automatically, e.g. by a computer?" We restrict ourselves to considering digital computers for which every message is finally coded by a finite sequence of bits. In practice this is no restriction at all, since other types of processing devices, e.g. analog computers, can be simulated easily by digital computers.

For a digital computer every message appears as a sequence of characters or other symbols. This statement is true as well for the input as for the output data. Hence the most general process which we have to consider is the transformation of strings of characters. All such transformations can be composed of elementary replacements of the form

(*) If a substring $a_1 \ldots a_{k-1}\ a_k\ a_{k+1} \ldots a_n$ occurs in the given string then it should be replaced by $a_1 \ldots a_{k-1}\ b\ a_{k+1} \ldots a_n$. If there are more such substrings then replace the leftmost one.

(The string b may be empty, i.e. a_k is deleted). In order to build constructively from such elementary replacements a procedure describing arbitrary processes, we must specify also the order of execution of these replacements and state when the processing ends. Since every replacement (*) requires a certain execution time we

cannot execute infinitely replacements if the process is to end at a certain time. Hence, all processes must consist of a finite number of steps: The description of such a process is called an algorithm (from the name of AL KHWARIZMI who lived in the 9th century. In fact the geometric constructions of EUCLID and his method for getting the greatest common divisor were already algorithms). If the algorithm is presented in a form suitable for execution by a computer we call it a program.

There are many distinct ways for describing algorithms. The use of replacements is only one of them. Already in the early days of computers it was recognized that not every algorithm needed to be decomposed into elementary replacements. It is sufficient to start from operations for which how to decompose them further is already known. For example, if one has shown that the usual addition and the other arithmetic operations may be done using elementary replacements, then one can use these operations in formulating algorithms; the operands of these operations are interpreted as numbers and not considered only as sequences of binary characters. These operations correspond to a higher level of abstraction in our hierarchy of interpretations as introduced in section 1.

Algorithms can also be constructed on a higher level of abstraction which is specifically developed for this purpose. The level is determined by first asking: "What are the basic operations and operands most suited to the problem to be solved? Once the algorithm is described in terms of these operations, one has the (simpler) problem of describing these operations and the data in terms of simpler operations and their data. This method is called construction of algorithms by step-wise refinement (cf.[Wirth 1971])

Algorithmic processes are not the only ways of deriving new informations, but they are the only ones which are constructive and lead effectively to the desired result after a finite amount of time. Examples of nonalgorithmic processes are all mathematical constructions which do not end after a finite number of steps, e.g., the computation of the decimal representation of π or other decimal fractions with infinitely many digits. Other examples include reasoning by analogy or by intuition. Even if it can be shown that some intuitive reasoning has an algorithmic basis complexity and the time required by such algorithms would mostly be so high that it is prohibitive for practical use.

Information processing by means of algorithms is thus restricted compared with information processing by human beings. Not all information processing by human beings can be taken over by computers. Expecially in medicine the class of algorithmically solvable problems seems at present to be rather small.

3. The relation of data and algorithms

Up to now we have looked at information processing systems from the point of view of a designer. It is this view which puts the main accent on algorithms and programs. From the point of view of the user of such a system we see mainly the data which we input or receive as results. The transformations which lead to these results remain mostly invisible. Nevertheless it is the algorithm to be performed which determines the amount of input data required, and the form and the order of these data. On the other hand we must not be satisfied by any coding, amount and order in which the resulting data are presented; it is usually not too difficult to modify an algorithm in such a way that it presents its results in an understandable form (e.g. as character strings or decimal numbers instead of sedecimal numbers), orders them in a proper fashion and, most importantly, delivers only those results which we are interested in.

These considerations lead to the conclusion that the order in which we design the input and output data and the algorithms is of some importance. It is necessary that we define first the required results, then outline the algorithms which lead to these results and finally determine the required input data. If we wish to use the input data for several applications, ideally we should proceed in this way for all applications. If this is impossible because some of these applications will be designed much later in time, we must still determine carefully what requirements may be imposed on the input data by these as yet unknown applications. All too often we see that the design is started first by outlining the input data, but the question should be: "What data do we need to derive these results?", not "What results can be derived from this data?". To think first of results and algorithms and only then of input data is of particular importance when we build information systems. An omission in entering patient histories into a computer probably can never corrected; even if we know the missing information the large number of records which must be corrected guarantees that the correction will never be done.

Also the form in which we store the data internally may be important. If we want to sort records in a file in order of increasing dates it is useful to have stored the dates in the format YY.MM.DD and not as they are usually written with the month or day first.

4. The external representation of programs and data

From now on we devote our discussion entirely to information processing using computers. One of the basic concerns is how to present algorithms (i. e. programs) and data in a way suited to both the human writers (and readers) and to the computer.

It should be clear from our remarks on how to present results that the form as bits, or a binary number, which is used inside a computer is not a suitable representation outside the computer. Instead we use representations which are more suited to human beings and which can be converted to the internal form for a computer by special programs in the computer itself. In the case of programs we call the set of rules which determine such an external representation a programming language. The converting program is called a compiler. In many cases the compiled program can be executed directly because it consists of instructions which are basic operations of the computer. If this is not the case (the compiled program is still on a higher level of abstraction) we need in addition an interpreter, a program which simulates the instructions in the compiled program with the help of the instruction set of the computer.

We distinguish three classes of programming languages:

- machine-oriented languages, in particular assembly languages. The basic operations of these languages coincide with the instructions (e.g. the use of mnmonics to denote operations or operands). There is usually one such language for each particular computer type.

- nigh level languages (also problem-oriented or procedure oriented languages). These languages allow, as far as numeric calculations are concerned, for a notation similar to standard mathematical notation. They furthermore contain means for expressing the structure of an algorithm and the flow of control in a neat and lucid way. Most well-known languages like FORTRAN, ALGOL 60, ALGOL 68, PL/I or PASCAL but also special language like SNOBOL 4, a language particulary suited to text processing, belong to this class.

- declarative languages (very high-level or problem-defining languages). High-level languages serve for formulating algorithms to solve a given problem. Declarative languages describe the problem; it is assumed that from this description one can automatically derive an algorithmic solution by selecting amongst well-known algorithms. Hence, a declarative language requires a good overall view of all important algorithms in a field of application, and a library of such algorithms ready for automatic adaption to a given problem. Declarative languages can thus be defined for relatively narrow fields of application only. They are the most convenient tool for users which one could think about. But practically speaking they are still of minor importance since the costs for developing and implementing such languages are still too high.

It is not our concern here to teach any of these languages. But in view of their importance we give a classifying overview of the

characteristic properties which one can find in most high-level languages. We take our examples from ALGOL 68 [Wijngaarden 69].

A program is a collection of statements describing actions to be executed on certain data. As the basis for structuring programs one finds in most newer languages blocks and procedures (fig. 1). A block is a parenthesized sequence of statements which if seen from the outside form a new (composite) statement. A procedure denotes a statement, or more generally, an algorithm which can be invoked at arbitrary points during program execution by just quoting its name.

The sequence of execution (flow of control) of such statements is controlled by a set of special statement types which we describe in figure 2.

For manipulating data one must know how these data are coded and how they are to be interpreted. This knowledge determines what operations are admissible We characterize this knowledge by the type or mode of the data (figure 3). Whenever we introduce a named data object we indicate its type. Besides the simple types we have type constructors for describing groups of data (arrays, records). Pointers serve to show the (dynamically changing) relations between objects. The most important operation which is explained for objects of all types is the assignment:

a:=formula

It allows for assigning the result of the formula to the variable a. By using the name a later on, this result may be retrieved (cf. [Goos 75] for a broader discussion of language properties).

There is a well-known relationship between a natural language and the thinking habits of the people using this language. The language mirrors the thinking habits of the people creating it. At the same time it forces people to think and to express themselves in the frame of this language. Ideas which cannot be expressed by simple means are likely not to be thought. Conversely ideas which can be expressed by simple means are considered to be simple even if they are of great complexity. The same arguments apply to programming languages, which in addition reflect, the structure of present-day computers and our understanding of what computers should do. Hence, the programming language influences at least the following:

- The conceptual understanding of how a problem can be solved by computers.
- The range of problems which can be attacked by programming.
- The set of basic notions available in programming.
- The style of programming (clarity, readability, modularity etc.).
- The reliability and correctness of a program.
- The meaning of efficiency.

Thus, the proper choice of a programming language may influence considerably the solution of a problem, the amount of time spent for design, coding, testing and maintenance and the overall costs of the project. This influence and its economic consequences are usually very difficult to estimate since rarely does one guide the same project twice just to obtain comparable figures (see [Goos 73] for a continuation of this discussion).

In an analogous fashion the proper design of input data for a program may influence considerably the reliability in use of this program. The very simple method of not coding everything numerically but having the computer look up the coding internally (e.g., that "wednesday" is coded by 3) avoids a lot of mistakes. An additional degree of reliability can be obtained easily by giving not only the values of the input data but also their meaning. The input line

age=43

is much preferable to simply giving the value 43, as far as convenience and reliability of usage are concerned.

5. The program development process

The development of a program may be subdivided into several steps (cf. [Metzger 73] for a similar scheme):

- system analysis and problem definition
- program design
- implementation
- system integration
- acceptance test and installation
- maintenance.

This section tries to characterize each such step by a few sentences. Each step forms a distinguishable phase of program development. They are not usually executed sequentially but can overlap partially in time. An important aspect of the development process is iteration: the return to an earlier phase, in particular the design phase, for improving and correcting shortcomings which are detected only in a later phase, or for adapting the program to changing requirements.

System analysis and problem definition

The goal of this step is the development of a model of the solution of the given problem, and the writing of a plan of how this solution

can be achieved. The model shows the feasibility of the proposed solution; if defines the problem in technical terms; it characterizes the necessary input data and the results to be obtained. Except that it proposes a certain decomposition of the problem into subtasks which are believed to be solvable, all specific decisions about algorithms etc. are deferred to the design phase.

A main concern of the problem definition must be to establish the criteria to be met by the solution, e.g., the class of computer configurations on which the program should run, whether the program should be portable, what are the trade- offs between reliability, date of delivery, efficiency etc. We discuss some of these criteria in more detail in the next section.

Program design

The goal of the design phase is a specification of the solution, its algorithms and data structures. For a large project the design is divided into a gross design specifying the subtasks which must be solved by some modules and the relationship between these modules, and the module design specifying the inner working of the modules. This subdivision corresponds to the principle of stepwise refinement mentioned earlier.

Design decisions are very often risky: they are based on incomplete information about their consequences. Whether the decision was good or not is seen sometimes only years later. Although the design itself may constitute only a small percentage of the overall costs of a project the costs of all later phases and the overall costs are implicitly determinded by the design. From an economics point of view the best people should therefore work on the design.

Whenever possible the design should include a rational - not an intuitive - reasoning about the correctness of the solution. It is much simpler and cheaper to correct mistakes and omissions now then in later phases.

Implementation

This phase codes and tests the solution developed in the design phase. It constitutes the last step in the refinement

gross design -→ module desgin -→ coding.

While creativity is a virtue of a designer the required property in implementation is utmost accuracy in all activities. Ideally, testing

is required only for removing the clerical errors introduced during the implementaion itself. One should always keep in mind Dijkstra's famous statement: "Program testing can be used to show the presence of bugs, but never to show their absence."

System integration

This step is done successively after the programmers deliver their tested modules. It is the attempt to put these modules together and to test the program as a whole. Usually this attempt fails in the beginning due to bad documentation of interfaces, misunderstandings, lack of communication about changes etc. At latest in this phase, the need to prepare from the beginning against misunderstandings, imprecision and other human inadequacies has been learnt.

Acceptance Test and Installation

This phase includes the thorough testing of the whole program by a group of people distinct from those who wrote the program. This acceptance test must show the functional correctness, the robustness and the performance of the program. It is a prerequisite to the installation of the program for practical use. At the same time it has to clarify whether the other prerequisites for installation, e.g., a user guide, program documentation etc. are in a satisfactory state. The group performing the acceptance test should design sets of test data with which the program has performed well and which can be used by customers to check that the program as installed performs well, at least in its main parts.

Maintenance

This phase consists of correcting detected errors, adapting to new environments (new hardware, new problem specification), and improving the performance of the program. It lasts for the whole life-time of the program. There are numerous examples that maintenance may incur more than half of the total costs of a project. It is therefore important to check again and again how long the maintenance should be extended and when it should be stopped in favour of a newly developed program which takes into account the lessons learnt from the last development.

6. Qualitative properties of programs

What constitutes a "good" program? We have mentioned a few properties of programs which are relevant in this context. This section is devoted to defining some of these properties and to discussing why they are relevant. We are not concerned about how to achieve these properties.

Functional Correctness (A)

The main requirement of a program is that, given an admissible set of input data, it precisely does what it is supposed to do. No other property of the program is of any concern if the program does not solve the specified problem. This requirement may be difficult to meet if the problem was not specified precisely enough and a proof that the progam is solving the problem is impossible. The situation should be carefully avoided by precise specification of the problem. Otherwise the result may surprise greatly both the programmer and his customer.

Being in Time (B)

This is another requirement which must be met if the problem is to be solved at all. It is not unusual for the programmers to exceed the date of delivery by continuously improving the performance of their program. Meanwhile the customer loses orders of much greater value than can ever be saved by the improvements.

Modularity (C)

A program is called modular [Dennis 73] if it consists of pieces, the program modules, such that

(a) the correctness of a program module can be demonstrated regardless of the context of its use in building larger units of software;

(b) program modules written under different authorities can be conveniently put together without knowledge of their inner working.

To be more precise this definition defines functional modularity. Other definitions, e.g., separate compilability are of minor significance. A sufficient modularization is a prerequisite for most of the program properties which follow.

Adaptability (D)

A program is adaptable to a new or changed problem specification if it can easily be modified to conform to the new requirements. An increase in efficiency usually diminishes the adaptability of a program.

Portability (D)

A program is portable if it can be easily transferred to a new base system, e.g., another type of computer, or another computer configuration with new peripheral devices. Hence, portability and adaptability are like the two sides of a coin. Portability is a prime requirement for all software house products. It should be a requirement of all adacemic software developments. For obvious reasons computer manufacturers have a minor interest in portability. Programs are claimed to be portable much more often than it is actually the case.

Compatibility (D)

Compatibility is the harmonious coexistence of several programs. It exists if we can use programs in combination (e.g., the use of FORTRAN subprograms in an ALGOL main program), or if two programs show similar behavior to the user and can be used alternatively. Compatibility is a primary requirement whenever one develops an existing system further or replaces it by a new one. Very often, however, the pursuit of compatibility leads to stagnation and prohibits the use of newer scientific insights.

Reliability (E)

Reliability comprises all properties which guarantee that the program is working properly. It thus includes functional correctness, i.e. safety against internal faults of the program, and fault tolerance, i.e. safety against faults in the base system, such a faults at the cpu, the main store, parity errors, faults of peripheral devices or break-down of electrical current. Ideally such faults should lead only to a drecrease in program performance and never to a complete break-down. Finally reliability includes robustness, i.e. safety against erroneous input data and wrong operating. Robustness requires that such errors should be detected at the earliest possible time and that recovery is made so that the program can continue, e.g., by requesting new input data.