NONPROCEDURAL COMMUNICATION BETWEEN USERS AND APPLICATION SOFTWARE

Bořivoj Melichar
International Institute for Applied Systems Analysis, Laxenburg, Austria

RR-81-22
October 1981

INTERNATIONAL INSTITUTE FOR APPLIED SYSTEMS ANALYSIS
Laxenburg, Austria

International Standard Book Number 3-7045-0016-X

Research Reports, which record research conducted at IIASA, are independently reviewed before publication. However, the views and opinions they express are not necessarily those of the Institute or the National Member Organizations that support it.

PREFACE

Bořivoj Melichar participated in the Junior Scientist Program held at the International Institute for Applied Systems Analysis during the summer of 1979, working with the Management and Technology Research Area in its study of the impact of small-scale computers on managerial activities. The difficulty of developing suitable software has been identified as one of the major obstacles to a healthy balance between cheap hardware and user-friendly software in this field. The use of nonprocedural languages is often suggested as one possible way of overcoming this problem, but as yet there has been no comprehensive survey of the subject. This report, written while the author was working at IIASA, represents one of the first attempts to compare different methods of nonprocedural communication between users and their application software in interactive data-processing systems.

Göran Fick
Management and
Technology Area

CONTENTS

NONPROCEDURAL COMMUNICATION BETWEEN USERS AND APPLICATION SOFTWARE

Bořivoj Melichar
International Institute for Applied Systems Analysis, Laxenburg, Austria

SUMMARY

This report is a survey of nonprocedural communication between users and application software in interactive data-processing systems. It includes a description of the main features of interactive systems, a classification of the potential users of application software, and a definition of the nonprocedural interface. Nonprocedural languages are classified into a number of broad groups and illustrated with examples. Finally, future trends in user–computer interfaces and possible developments in manager-oriented languages are discussed.

1 INTRODUCTION

1.1 Basic Concepts

Advances in semiconductor technology during the past decade have dramatically increased the availability of low-cost computer hardware. One of the results of this greater availability has been the development of cheap but powerful small-scale computer systems.

According to Fick (1980), the power of computer systems has recently been doubling every two years, while the price of computer systems has remained approximately constant. With the "real" cost of computing capability declining, it is nevertheless apparent that low-cost computer hardware does not necessarily mean "cheap" computing – the cost of the software should also be considered. Computer software is a labor-intensive product, generally designed specifically for a small group of users or even for an individual (Fick 1980), and it is therefore more expensive than the mass-produced hardware. There are a number of areas in which development should take place if low-cost hardware is to be matched with suitable software, and these areas are outlined below.

1. Development of theoretical and methodological tools for software design in different fields of application.

2. Development of tools for software realization (programming languages, automatic program generation, program debugging and verification, etc.).
3. Development and production of media for software distribution (semiconductor read-only memories, magnetic tapes, magnetic discs, punched cards, punched paper tapes, books, journals, etc.).
4. Development of means for communication between users and the application software (input/output devices, new languages, etc.).

In this article we survey problems of communication between users and their application software. The manner of communication between users and application software is highly dependent on the users' access to the computer system. In recent years there has been much discussion of the issue of "indirect access" vs. "direct access", e.g., batch-processing systems vs. time-sharing systems. The communication between user and computer is very slow in a batch-processing system; the user can neither influence the way in which the program is run nor intervene while it is being run. The issue of batch-processing systems vs. time-sharing systems has therefore been resolved in favor of time-sharing systems. This means that communication between user and computer is now generally interactive in nature. The rest of this paper assumes the use of interactive systems and discusses the most interesting features of these systems.

1.2 Interactive Systems

Many of the problems associated with batch-processing systems may be overcome by the opportunity to communicate directly with the computer using an interactive system. However, the use of interactive systems has helped to create various new problems, which are now receiving considerable attention; the main requirements of interactive systems and the basic principles of the interactive dialogue are still under debate (Miller and Thomas 1977, Watson 1976, Fitter 1979, Gaines and Facey 1976). Here we provide a list of facilities which we think could and should be provided by interactive systems and some principles which should be followed in an interactive dialogue. Some of the following points may also be relevant to a batch-processing (noninteractive) environment, but we consider only their importance in interactive systems.

System response time. System response time is the time spent in processing the input and in producing a response. It is difficult to know exactly how long or short the response time should be, and there is no general agreement on this subject. There are several arguments for short response times:

- human response times are of the order of two seconds
- long response times decrease throughput
- long delays are usually disruptive and disturbing

On the other hand, there are arguments against short response times:

- short response times require high investment in the system
- long response times might be helpful for more complex tasks
- fast responses may encourage users to expect the same level of service at all times

It has been observed that the variability of the system response time can be very annoying to the user.

Rohlfs (1979) proposed that systems should be designed so that their response time may be adjusted to the activity of the individual user:

> 15 sec intolerable
> 4 sec too long in most cases, possibly tolerable after termination of a major step
> 2 sec too long for very involved work
< 2 sec necessary for work consisting of more than one step
< 1 sec immediate reaction

Availability and reliability. The computer system should be available for use at any time; this would be possible if each user had his/her own computer. Because the user will be unhappy with any system performance error or degradation regardless of good normal performance, reliability is also a very important feature of an interactive system. For many computer applications almost no degradation or loss in availability can be tolerated.

Commonality. A software system is usually composed of a number of subsystems. In this case, the system should be organized such that terminology does not change between subsystems. This implies that the input language of each subsystem should be an extension of the common base language. Thus, the user will only need to learn additional functions or statements when using a new subsystem and not have to learn a completely new "foreign" language. When the user is in trouble, he/she can use standardized "help" functions to extricate him/herself from the situation.

Adaptability to user proficiency. It should be possible to design the interface between user and computer to suit users with different amounts of knowledge about a particular subsystem. A sophisticated user may prefer to use mathematical or formalized notation in his/her dialogue. On the other hand, a novice user is likely to prefer less formalized notation and use simpler system functions.

The newer systems have been adapted for users with various levels of proficiency by designing different user interfaces. As the user becomes more proficient, he/she can use more sophisticated functions or a more formalized interface.

Immediate feedback. A system should make an unambiguous response to each of the user's requests. This response should be sufficient to identify the activity and state of the system. In situations where system response times are longer than usual, it is highly desirable to confirm receipt of the user's command immediately. It is very useful to let the user know when the computer will produce a response, for example, by displaying a countdown clock on the terminal.

Observability and controlability. A system can be regarded as an automaton. It is important that the user should feel in control of the system, and in order to make this control possible he/she must have some knowledge about the current state of the system. Thus, when the user's input is processed, the user should be informed about the current state of the system. The display of this information may be regarded as a transition from one state to another.

Use the user's model. Everybody rationalizes their experiences in their own terms, and in the same way each user will model a computer system according to his/her experience of it. This cannot be prevented and should be made as easy as possible. The system

should use a model of computer activity which corresponds to that perceived by the user, so that the interactive dialogue resembles a conversation between two users accepting the same model. Given that we can somehow determine the user's model of the computer system, we should make the underlying processes reflect it, and design the dialogue to reveal it as clearly as possible.

Validation. All input commands and data must be validated by checking syntax, semantics, and, if possible, values. The system must inform the user about any errors or ambiguities in the input data and let the user update the values in question before the system acts upon them.

Query-in-depth. Information and advice on the system should be categorized according to possible user requests and should be available to the user through a simple standard mechanism.

Extensibility. There will never be enough professional programmers and system developers to provide all the tools that users may desire for their work. It should therefore be possible for users to develop new tools or to extend the functions already present in the system.

Written documentation. In some cases it is necessary to produce high-quality documents as a result of interaction between user and terminal. Text processing is one of the most important examples.

System activities. It is necessary to maintain records of system performance and user's activities to evaluate and improve the behavior of the system.

1.3 Interface between the User and the Application Software

The nature of a user–application software interface is largely determined by the medium used for communication. The most basic means of communication are alphanumeric texts and graphics. More advanced methods of communication, such as speech, eye movement, brain-wave control, and handwritten script (Watson 1976) are currently being investigated.

In this survey we concentrate on alphanumeric texts as a medium for user–computer communication. It is assumed that a normal keyboard and alphanumeric display (with or without hard copy) are available to the user.

The user–software interface has two sides (Watson 1976, Sprague 1981): the input side, through which the user inputs information by means of an *action language*; and the output side, through which the computer provides feedback and other assistance to the user by means of a *display language*.

Let us first survey the action languages. A wide range of action languages has been designed to accommodate a wide variety of users. The selection of a particular action language determines the communication mode that should be used. We can classify action languages and/or communication modes as follows:

- low-level machine-like programming languages
- high-level universal programming languages
- high-level programming languages with new syntax and semantic forms (such languages can be used as special-purpose languages)

- self-contained special-purpose languages
- answer languages
- command languages
- query languages
- natural languages
- two-dimensional positional languages

This classification of communication modes covers the complete range from artificial machine-oriented languages to natural human-oriented languages.

Different types of user may wish to communicate with the computer in different ways, and so it is very important during software development and implementation to select the communication modes appropriate to the end-users.

According to Schneiderman (1978), we can categorize users into three groups with respect to their skills, the frequency with which they use application software, their degree of knowledge of the problem under investigation, and their professional fields.

1. *Nontrained intermittent users* who infrequently use application software. A user in this category is called a "casual user" by Codd (1977) and a "general user" by Miller and Thomas (1977). These people are not computer professionals, have no syntactic knowledge, and have little knowledge of the organization of the application software. At the same time it is assumed, however, that these users have sound professional knowledge in their own particular fields and, therefore, that the system should allow them to express themselves in their own specialized terminology (Lehmann 1978).
2. *Skilled frequent users* who make daily use of application software. These users can learn the simple syntax of a communication action language, but they are more interested in their own work than in computer programming. This category includes skilled secretaries, engineers, and managers.
3. *Professional users* whose main task is to develop and maintain the application software. They are highly trained in this field and are concerned with the efficiency and the quality of the computer system. This category includes database administrators and software system programmers.

Although programming and communicating with a computer in high-level programming languages like ALGOL, FORTRAN, COBOL, PL/1, and PASCAL represents a major advance over the use of machine-like low-level programming languages, high-level languages are becoming less appropriate now that cheap hardware is available. With the rapid diffusion of cheap computer hardware it is expected that the numbers of people in the nontrained intermittent and the skilled frequent user categories will grow very quickly. These users have little or no experience of data-processing, and would find it very difficult and very time-consuming to learn how an algorithm may be constructed and described in a programming procedure-oriented language. Therefore, it is highly desirable to find some means by which these users may communicate with application software in a nonprocedural manner.

There is very great interest in the development of nonprocedural languages, not only to facilitate communication between the user and the application software, but also in connection with the implementation of application software.

According to Winograd (1979), it is useful to distinguish three ways in which computational processes may be specified:

1. *Program specification.* A formal specification which can be interpreted as a set of instructions for a given computer. This is the imperative mode characteristic of traditional procedure-oriented programming languages.
2. *Result specification.* A process-independent specification of the relationship between the inputs (or initial state), internal variables, and outputs (or final state) of the program.
3. *Behavior specification.* A formal description of the activity of a computer over time. A description of this type selects certain features of the machine state and action without specifying in full the mechanisms which generate them.

We can divide an algorithm into two components (Kowalski 1979), a *logic component*, which specifies the knowledge to be used in solving the problem, and a *control component*, which determines the way in which the knowledge will be used. For example, consider an algorithm for computing factorials. The logic component of the algorithm is given by the definition of a factorial:

```
1 is the factorial of 0

u is the factorial of x if v is the factorial of
  x - 1 and
u is v times x
```

This is an example of result specification. For comparison, consider the following procedure in ALGOL 60:

```
procedure factorial (x); value x; integer x;
if x = 0 then factorial := 1 else
              factorial := factorial (x - 1)*x
```

In this procedure the logic component is blended with the control component.

According to McCracken (1978), we can characterize nonprocedural languages in two ways.

1. The user cannot control the storage of data. Decisions that relate only indirectly to the calculation are considered to be part of the internal functioning of the system. These include decisions about the internal representation of numbers (fixed point, floating point, octal, decimal), dimensions of quantities that occur only as intermediate results, and input and output formats. The representation of data is selected by the system itself, and the description of the data representation is stored with the data. This is called *data independence*.
2. The user cannot tell the computer *how* to obtain the desired results. Rather, he/she tells the computer only *what* he/she wants. This means that the user does not become involved in the loops and branches which make up most of the computational steps in a program written in procedural language. This we can call *control independence*.

The following query on the data stored in a data base provides a nice illustration of the nonprocedural approach.

```
RETRIEVE (AGE > 40 AND < 65) AND SALARY >3000;
FOR EACH
      IF WEIGHT > TABLE (HEIGHT - 50)
      THEN SET OVERWEIGHT = "TRUE"
      PENSION = SALARY/3;
      ELSE SET PENSION = SALARY/2
```

We can now give a working definition of a nonprocedural language:

> In a nonprocedural language the computational process is specified by the desired result (or behavior). This specification is data independent and control independent.

We shall consider a nonprocedurally *oriented* language to be a language which does not fulfill all of the conditions necessary for classification as a nonprocedural language, but which does not require program specification. Of the nine communication modes listed earlier, we can consider answer languages, command languages, query languages, natural languages, two-dimensional positional languages, and some special-purpose languages as nonprocedurally oriented action languages.

In the next section we examine the nonprocedural action languages available for communication between nontrained intermittent or skilled frequent users and application

software. Communication in the reverse direction (from the system to the user) takes place through display languages, the main features of which are described in Section 3.

2 NONPROCEDURALLY ORIENTED ACTION LANGUAGES

In the last section we defined nonprocedural action languages and sketched the arguments for using them for user–application software communication. In this section we shall give the basic characteristics of each type of language and illustrate them with examples taken from the literature.

2.1 Answer Languages

An answer language is the set of words, phrases, or sentences which may be used to answer questions asked by the computer. This type of language is introduced first because it is the simplest means of user–computer communication.

The answer languages used as action languages are very closely related to display languages. The display language in this case contains, among other things, the set of questions which are asked and which the user is obliged to answer. We can categorize answer languages with respect to the number of different questions that can be answered in each user response: this may be one, or more than one. Languages in which the user may answer only one question at a time can be divided into three classes: binary answer systems, menu selection systems, and instruction and response systems.

Binary answer systems only recognize two answers, YES and NO, often represented by the abbreviations Y and N. The binary answer languages are used in software systems in which the internal structure corresponds to a binary oriented graph. In the binary oriented graph two branches leave each edge. Edges correspond to states of the system; in each such state the system asks a question, and according to the answer (NO or YES) a branch is selected which leads to a new state. The binary answer language is used mostly in simple systems such as computer games.

As an example we use the popular game blackjack (Thompson and Ritchie 1975). The dealer (simulated by the computer) might ask the following questions:

```
NEW GAME?
HIT?
INSURANCE?
SPLIT PAIR?
DOUBLE DOWN?
```

Each question is answered by YES or NO.

It is clear that binary answer language may be used only in systems in which a limited number of questions may be asked. For cases in which the answers YES or NO are not sufficient to answer all the questions which may arise, we can use a *menu selection system* (*n*-ary answer language).

In a menu selection system the set of possible answers to each question is defined. Each set must be finite, and from a practical point of view should consist of only a small number of answers.

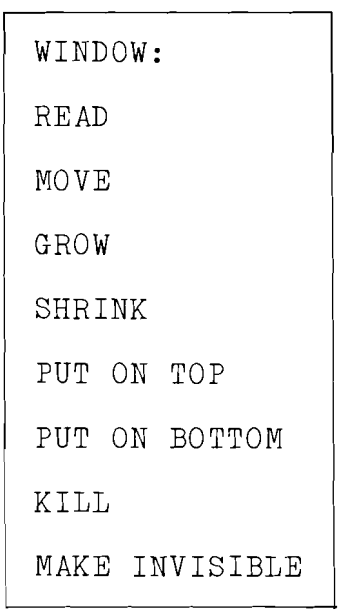
The set of answers to a particular question is called the "menu". There are two ways in which the menu may be presented to the user. First, the menu of answers for each question may be given in the description of the software system, and the user is thus obliged to learn these menus before using the system. Much better is the second way, in which the menu and the question are provided together. This method has a number of advantages, the most important of which is that the user need not learn the menus for all possible questions.

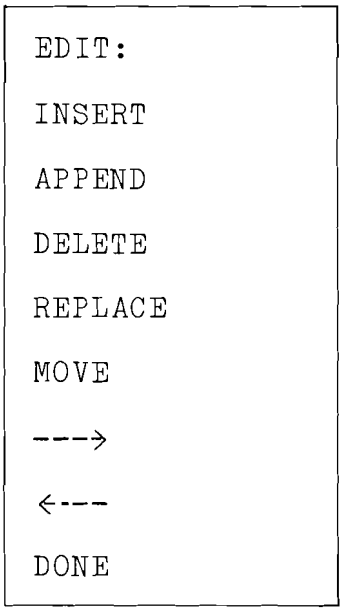
Menu selection systems, like binary answer languages, are used in software systems in which the internal structure corresponds to an ordinary oriented graph. In such a graph a varying number of branches leave from each edge, the edge representing the question and the branches corresponding to the set or "menu" of possible answers.

As an example we use some menus from Teitelman (1979), who describes a system designed to help the user to develop programs. In this system, for example, the user may be presented with the following choice:

```
MENUS:
WINDOW
DOCUMENT
EDIT
LOOK
HISTORY
BREAK
OPERATIONS
```

This menu is then used to select further menus.





Questions with menus are displayed on a screen, and the user can select an answer by means of the cursor.

When the number of possible answers to a particular question is very large, it is inefficient (or sometimes impossible) to display all possible answers with the question. This situation may arise if the answer contains a number. In such cases we may use an *instruction and response system*.

In an instruction and response system an explanation of the answer required is included in the question. The following example of an instruction and response dialogue is taken from Hebditch (1979).

```
ORDER OR CREDIT? O
CUSTOMER NUMBER? 848923
CUSTOMER IS BROWN'S STATIONERS LTD.
            HIGH STREET
            WATFORD
PLEASE CONFIRM (Y/N)? Y
DELIVERY ADDRESS IS AS ABOVE
PLEASE CONFIRM (Y/N)? Y
```

```
ORDER NUMBER? 77/34
DISCOUNT? 12.5
***12.5 PERCENT IS HIGHER THAN NORMAL TERMS
PLEASE CONFIRM BY RE-ENTRY? 12.5
ENTER PRODUCT CODE. QUANTITY (END AFTER
LAST ITEM)
?B04,24  24 DOZ PENCILS (HB)
?A68,10  10 REAMS BANK PAPER
?B61,36  36 DOZ BALL-POINT PENS
***36 DOZ IS ABNORMAL QUANTITY FOR THIS ITEM.
PLEASE CONFIRM
?B61,3  3 DOZ BALL-POINT PENS (BLUE)
?Z15,1  1 DISPLAY STAND (BALL-POINT PENS)
?END    ORDER COMPLETED (4 ITEMS)
DO YOU WISH TO SEE INVOICE PRIOR TO PRINTING
(Y/N)? N
```

Systems in which a user can answer more than one question at a time require some type of fixed format user input. This format can be used, for example, to separate single answers in user input. and may be described in the question. Such a system is called a *displayed format system*.

The following simple and self-explanatory example of a book-ordering system (Hebditch 1979) shows a question containing the format description and the resulting answer.

```
ENTER AUTHOR / TITLE / PUBLISHER / ISBN /
      NO. OF COPIES / CUSTOMER NAME /
      CUSTOMER ADDRESS / POST OR COLLECT?
```

```
HEBDITCH / DATA COMMUNICATIONS: AN
INTRODUCTORY GUIDE / ELEK SCIENCE LTD. /
NK / 4 / A. WISEMAN / NA / COLLECT
```

2.2 Command Languages

Command languages in one form or another have been in use since the earliest operating systems first came into existence in the late 1950s. The name "command languages" was used in the past to describe the job control languages used as interfaces between users and operating systems. Today the term includes a wide variety of languages employed as user–computer interfaces in many types of software systems.

A command language consists of a set of commands. A typical command is composed of four elements: the command prefix, the operation specification, the parameter part, and the command completion.

The first part of the command, the *command prefix*, contains

- a command indication (a symbol or string of symbols to distinguish the command from other inputs)
- a command identification (a label or number used for reference purposes in other commands)
- a condition, which must be satisfied before the command is executed (for example: `IF TIME < MAXTIME THEN`)

A command word is frequently reserved for use as an *operation specification* (Miller and Thomas 1977). Watson (1976) proposes an operation specification in the form of a verb–noun pair. In this case we obtain a verb–noun matrix as, for example, in an editing system:

	character	line	page
delete			
insert			
change			
move			

Each element of this matrix is a normal English command.

Displaying the operation specification in this form seems to be very helpful for users with no experience in data processing (Keen and Hackathorn 1979). Hebditch (1973) proposes a more structured format, using adjectives (J) in addition to nouns (N) and verbs (V), to create an operation specification with three forms:

```
V J N      (PRINT CONDENSED RECORD)

V N J      (FIND EMPLOYEES WITH A DEGREE)

V J N J    (PRINT ALL LINES BEGINNING WITH +)
```

Further, Hebditch (1973) proposes that a set of basic verbs could be used as an interface with a data base, as shown in Table 1.

TABLE 1 Computer functions, verbs commonly used to specify the function, and alternative specifications.

Function	Verb (abbreviation)	Alternative forms
Initiation	START (S)	Begin, Sign-on, Initiate, Go, Set-up, Evoke
Location of logical record	FIND (F)	Locate, Get, Search, Read, Obtain, Pick (good for inventory data base?)
Display of data item	PRINT (P) (for hard copy)	Display (for VDUs), Show, Query, Give, List, Present
Amendment of data item	ALTER (A)	Change, Amend, Modify, Replace, Convert, Set
Addition of new record or item	INSERT (I)	Add, New, Assign, Include, Originate, Form
Movement of record (or data) from one logical location to another	MOVE (M)	Transfer, Shift, Relocate, Convey, Reallocate, Transpose
Obtain assistance	EXPLAIN (E)	Assist, Why?, Expand, Clarify, Help, Interpret
Termination	HALT (H)	End, Finish, Done, Close, Terminate, Conclude

The *parameter part* specifies the operand and suggests various ways in which the command could be executed. There are two distinct methods of formatting the arguments for commands: *positional format* and *keyword format*.

In the positional format, particular pieces of data must appear in fixed relative or absolute positions in the parameter part.

In the keyword format the parameter part is a permutable string of arguments, each argument containing a *keyword* which indicates the argument type and, sometimes, its value.

Both types of argument format occur in current systems. From the user's point of view the keyword format is more acceptable, because it requires the user to memorize less information.

The arguments in the parameter part may be composed of several different elements; these may include keywords, constants (numerical, boolean, etc.), text strings, and expressions (regular, arithmetical, boolean, etc.).

There remains the question of what to do when the user does not specify some information that either could or should have been provided. There are several ways of prompting the user for missing information:

1. Listing the missing argument names with all their possible values so that the user may choose the correct values.
2. Assigning a *default* value automatically to some of the missing arguments and asking the user for agreement.
3. Supplying missing information on the basis of the arguments provided to previous commands.

The problem of choosing argument delimiters is related to argument specification. Delimiting functions may be delimiting command words, delimiting arguments, or delimiting optional arguments (arguments with default values) (Watson 1976). It is generally advisable to use the same symbol for delimiting command words and arguments and to use a different symbol for delimiting optional or default arguments.

The last part of the command is the *command completion*. According to Watson (1976), there are three types of command completion:

1. Command accept: a command completion indicating that the command should be executed and the system should then return to the base state to receive the next command.
2. Repeat: a command completion indicating that the command should be executed and the system should then return to an intermediate command state for quick repetition of the command with or without request. This mode is useful when an operation must be repeated several times.
3. Insert: a command completion indicating that the command should be executed, the system should then enter insert-command mode for insertion of some new parameters, and then the command should be repeated.

A different symbol should be used for each type of command completion.

2.3 Query Languages

According to Olle (1973), there are four levels on which a user might come into contact with a data base. At the highest level is the data administrator, that is, the person responsible for the data base. The applications programmer occupies the second level, and at the next level is the application specialist, who is able to formulate questions about the data stored in the data base but is not a programmer. Finally, people who are unable to formulate questions occupy the lowest level in this hierarchy.

Data administrators and applications programmers generally use programming languages in their work. Users who are unable to formulate questions may use the simpler answer languages discussed previously. Query languages are designed to be used by the intermediate group of people, users who are not programmers but who understand how to formulate questions for a particular application.

Query languages are high-level nonprocedural data-base languages, which allow the user to perform operations such as insertion, deletion, and retrieval on the data base. Strong emphasis is placed on retrieval operations and, in view of this fact, a finer categorization of retrieval operations seems appropriate (Schneiderman 1978). There are four main retrieval operations which may be performed on a data base:

1. Simple verification of the presence, absence, or acceptability of a specified item.
2. Retrieval of a single record when a key is provided.
3. Retrieval of a number of records when a key or boolean predicate is provided.
4. Total listing of all information stored.

The list below indicates how query languages may be used to sort and retrieve data, and gives some examples of the type of queries which may be asked (Schneiderman 1978).

1. Simple mapping produces data values from one field when a known value for another field is supplied. Example: Find the names of all employees in department 50.
2. It is possible to select all of the data associated with a specified key. Example: Give the entire record for the employee whose name is John Jones.
3. In a relational model it is possible to select any domain of a relation. Example: Print the names of all employees.
4. Boolean queries permit AND/OR/NOT connections. Example: Find the names of those employees who work for Smith and who are not employed in department 50.
5. Set operation queries involve set operations such as intersection, union, and symmetric difference. Example: Find the names of the employees who work for Smith and the addresses of the employees who work for Black.
6. Built-in functions such as MAXIMUM, MINIMUM, AVERAGE, SUM, make it easier for the user to formulate questions. Example: Print the sum of salaries of employees in department 50.
7. Combination queries are produced by using the output of one query as the input for another. Example: Find the names of all departments which have more than 30 employees and then find the names of the department managers.

8. It is possible to group items with a common domain value. Example: Print the names of departments in which the average salary is greater than $15,000.
9. Universal quantification corresponding to the "for all" (∀) concept of the first-order predicate calculus. Example: Find the addresses for all employees.

The features listed above are available in most query languages designed for data bases using relational, hierarchical, or network models of data.

As an example, consider the following data base, which is built on a relational model (Chamberlin 1976):

PRESIDENTS

NAME	PARTY	HOME-STATE
Eisenhower	Republican	Texas
Kennedy	Democrat	Massachusetts
Johnson	Democrat	Texas
Nixon	Republican	California
Carter	Democrat	Georgia
Reagan	Republican	California

This relation PRESIDENTS has domains NAME, PARTY, and HOME-STATE.

ELECTIONS-WON

YEAR	WINNER-NAME
1956	Eisenhower
1960	Kennedy
1964	Johnson
1968	Nixon
1972	Nixon
1976	Carter
1980	Reagan

This relation ELECTIONS-WON has domains YEAR and WINNER-NAME. The two relations PRESIDENTS and ELECTIONS-WON are the only relations in our sample data base.

According to Chamberlin (1976), there are four classes of query language: relational calculus-based languages; relational algebra-based languages; mapping-oriented languages; and graphics-oriented languages. Languages in the first three categories may be distinguished by their mathematical basis. The fourth category includes certain two-dimensional languages.

One example of a *relational calculus-based* query language is the language QUEL (Stonebraker *et al.* 1976). A typical query in QUEL has three parts:

- a *result name*, which is the name of the relation from which data will be retrieved
- a *target list*, which specifies the particular domains of the relation from which data will be retrieved
- a *qualification*, which specifies certain conditions that the retrieved data must fulfill

A QUEL interaction must include at least one RANGE statement to specify the relation over which each variable ranges. Two examples of queries in QUEL are given below.

1. What was the home state of President Kennedy?

```
RANGE OF P IS PRESIDENTS
RETRIEVE INTO X (STATE = P.HOME-STATE)
WHERE P.NAME = "KENNEDY"
```

2. List the years in which a Republican from Illinois was elected President!

```
RANGE OF E IS ELECTIONS-WON
RANGE OF P IS PRESIDENTS
RETRIEVE INTO Y (YEARS = E.YEAR)
WHERE P.PARTY = "REPUBLICAN"
  AND P.HOME-STATE = "ILLINOIS"
  AND P.NAME = WINNER-NAME
```

Relational algebra-based query languages use a variety of operators that deal with relations, yielding new relations as a result. Among the most important of these operators are projection, restriction, and set-theory (union, intersection, and symmetric difference) operators. Translating the two queries given above into relational algebra-based query language we obtain:

1. What was the home state of President Kennedy?

```
PRESIDENTS [NAME = "KENNEDY"] [HOME-STATE]
```

The above example uses projection and restriction operators.

2. List the years in which a Republican from Illinois was elected President!

```
(ELECTIONS-WON [WINNER-NAME = NAME] PRESIDENTS)
[PARTY = "REPUBLICAN"] [HOME-STATE =
"ILLINOIS"] [YEAR]
```

In this example we use union, projection, and restriction operators.

The basis of *mapping-oriented* query languages is the operation of "mapping", in which a known domain or set of domains is "mapped" into a desired domain or set of domains by means of some relation. Our two examples are now in the mapping-oriented language SEQUEL (Astrahan *et al.* 1976).

1. What was the home state of President Kennedy?

```
SELECT HOME-STATE
FROM PRESIDENTS
WHERE NAME = "KENNEDY"
```

2. List the years in which a Republican from Illinois was elected President!

```
SELECT YEAR
FROM ELECTIONS-WON
WHERE WINNER-NAME =
      SELECT NAME
      FROM PRESIDENTS
      WHERE PARTY = "REPUBLICAN"
      AND HOME-STATE = "ILLINOIS"
```

Graphics-oriented query languages are mentioned later in this survey in the section dealing with two-dimensional positional languages.

The mapping- and graphics-oriented query languages are designed for users with no experience in data-processing and offer power equivalent to relational algebra- or relational calculus-based languages while avoiding difficult mathematical concepts.

2.4 Natural Languages

The idea of communicating with computers using a natural language has provoked much discussion from the early years of machine translation. However, though this concept is obviously very attractive to the user, the implementation of a natural language interface presents considerable difficulties to the programmer.

Natural language is the technique of verbal communication between people. According to Addis (1977), natural languages have an extremely complex structure because they reflect the way in which people think.

The use of a natural language for user–computer communication has several major advantages (Infotech International Ltd. 1979).

1. It provides a familiar way of forming questions. This means that the natural language interface would be available to a large number of users without the need for special training.
2. There are often many ways to extract the same data. The user can usually communicate his/her knowledge in a natural language augmented by specific notation and vocabulary characteristic of his/her specialist field.
3. It may be easier to formulate complicated queries using a natural language than using formal languages or menu selection methods.
4. The user does not have to learn a formal syntax and his/her departures from accepted grammar may be tolerated without comment.

At the same time we must note the following disadvantages:

1. The use of a natural language interface encourages an unrealistically high expectation of system power.
2. The linguistic limitations of such a system are not as well-defined as they are within a formal language. Confusion can arise as the result of an unknown word, an unknown grammatical construction, or a misunderstanding.
3. Sentences in natural languages are frequently ambiguous. Implementation is difficult if all the possible meanings of a sentence must be considered.
4. Because much of the vocabulary may be specific to the particular application, the system has to be partially recast to deal with each new application.
5. A system with a natural language interface is inherently much more complicated to implement than systems using more formal languages.

Besides these disadvantages there is one additional problem: the great variety of natural languages in the world. These problems do not imply that a natural language interface is useless, only that it is not as generally applicable as is sometimes thought.

Natural language communication with computers falls into two major categories (Kraegeloh and Lockemann 1978): communication with systems designed to understand natural language as part of the development of artificial intelligence, and communication with data-base systems.

In developing artificial intelligence the main goal is for the system to understand a natural language by the simulation of natural language perception. The resulting systems are highly complex, because the semantics of the natural language must remain unrestricted. One of the characteristics of Man is his ability to produce a mental image of his environment, a model of the real world. During a conversation, statements about his environment are related to this cognitive model and can modify it. Simulation of this process requires some system in which the real world, or part of it, may be modeled in a computer.

One system capable of understanding a natural language is GUS (Genial Understander System), which is described in more detail by Bobrow (1977). This system is restricted to the role of a travel agent in a conversation with a client who wants to make a trip to a single city in California. The following is a simple dialogue between a client and GUS.

```
GUS:     Hello. My name is GUS. I can help you
         plan a simple trip by air. Where do
         you want to go?
Client:  I want to go to San Diego on May 28.
GUS:     What time do you want to leave?
Client:  I must be in San Diego before 10 am.
```

GUS: Would you like P.S.A. flight 102 that arrives at 9.15 am?

Client: What is the next flight?

GUS: Air California flight 310 that leaves at 8.30 am and arrives at 10.00 am.

Client: I'll take the first one.

GUS: In what name should I make a reservation, please?

Client: Dan Bobrow.

GUS: I have confirmed the following flight: P.S.A. flight 102 on Wednesday May 28 from San Jose to San Diego that leaves at 7.30 am and arrives at 9.15 am. What date do you want to return on?

Client: On Friday in the evening.

GUS: Would you like the flight that leaves at 7.45 pm?

Client: That's fine.

GUS: I have confirmed the following flight: P.S.A. flight 307 on Friday May 30 from San Diego to San Jose that leaves at 7.45 pm and arrives at 9.30 pm. Thank you for calling. Goodbye.

The second important area in natural language communication with computers is the use of natural languages to provide an interface between the user and data-base systems. This is a subject under much discussion because it could provide easy data access to users unfamiliar with query languages. In this case the computer's understanding of a natural language is not based on a simulation of the cognitive process; the data base itself serves as a model for understanding. Consequently, the input language should be restricted to the vocabulary necessary to use the data base. Under these conditions the natural languages used as interface languages in data-base systems may still be considered formal languages.

As an example of a natural language interface between user and data base we use the system PLANES (Waltz 1978). This system was developed with the following aims:

1. To accept the user's natural English
2. To provide explicit, clear answers to questions (possibly using graphics)
3. To tolerate minor errors (such as misspelling and grammatical errors)
4. To use clarifying dialogue for several purposes:
   - to increase its understanding of the user's request
   - to ask the user about parts of the request which it does not understand
   - to add new words, phrases, and sentences to its vocabulary
   - to provide appropriate assistance to the user
   - to provide information about the capabilities of the system
5. To be convenient to use, i.e.,
   - to be on-line and interactive
   - to operate rapidly
   - to require a minimum of typing
   - to answer a substantial percentage of questions
6. To be relatively easy to develop further

An ordinary dialogue between a user and the PLANES system is given below.

```
PLEASE ENTER YOUR QUESTION.....................

>>How many flights did plane 3 make in Jan 73?
```

PLANES searches the MONTHLY FLIGHT AND MAINTENANCE SUMMARIES and paraphrases the answer required as the value of TOTAL FLIGHTS for plane SERIAL #3 during January 1973.

```
SHOULD I EVALUATE THE QUERY? TYPE Y OR N
>>y
EVALUATING.....
(SUM TOTFLTS) = 17
PLEASE ENTER YOUR QUESTION.....
```

For our second example we give a clarifying dialogue involving a spelling correction.

```
>>Which Skyhawks logged less than 10 hours?
PLEASE ENTER A SPECIFIC TIME PERIOD
>>Between Jan. 1 and Jule 1 '73
I DON'T KNOW THE MEANING OF JULE.
PERHAPS IT'S MISSPELLED?
PLEASE ENTER A NUMBER CORRESPONDING TO THE
FOLLOWING:
1.   JUNE
2.   JULY
3.   NONE OF THE ABOVE
>>2.
O.K.
```

A natural language may not provide the best interface in every situation. Schneiderman (1978) describes a "natural versus artificial query language experiment" concerning communication with data bases and concludes that the user must have some knowledge

of the application domain if a natural language interface is to be used; i.e., the user has to be familiar with the semantics of the information in the data base. When a user learns a query language, he/she automatically learns the semantics of the information stored in the data base at the same time.

2.5 Special-Purpose Languages

In some fields, specific formalisms are used to describe particular problems. It seems reasonable to use these formalisms directly as *special-purpose languages* to interface with specialized software systems.

As an example, we consider one class of formalisms used widely for language design and implementation. Special-purpose languages based on formalisms of this type are used as interfaces in written translation systems. These languages are based on the idea of a context-free grammar, i.e., a set of rules of the form:

left part : right part

where the *left part* is one nonterminal symbol called a syntactic category and the *right part* is a string of nonterminal and terminal symbols.

The way in which sentences, composed of terminal symbols, may be derived from one particular nonterminal symbol, known as the start symbol, is first defined. The set of all sentences which can be derived from the start symbol may be described as a formal language generated by the grammar.

The following example illustrates the use of a language based on context-free grammar. In this case nonterminal symbols are represented by mnemonic names between angular brackets ⟨ ⟩; the terminal symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and /. The rules are:

```
<date>   : <number>/<number>/<number>
<number> : <digit>
<number> : <number>/<digit>
<digit>  : 1
<digit>  : 2
<digit>  : 3
           .
           .
           .
<digit>  : 9
<digit>  : 10
```

The start symbol is <date>, and we shall use the symbol → to represent one step in the derivation.

```
<date> ——> <number>/<number>/<number>
       ——> <digit> /<number>/<number>
       ——>    1    /<number>/<number>
       ——>    1    /<digit> /<number>
       ——>    1    /   2    /<number>
       ——>    1    /   2    /<number><digit>
       ——>    1    /   2    /<digit><digit>
       ——>    1    /   2    /   7<digit>
       ——>    1    /   2    /   79
```

The language generated by this grammar is a set of sentences of the form number/number/number/, which can be read as dates.

Context-free grammars are often used to describe the syntax of formal and natural languages and, as mentioned above, they can also be used as a basis for the special-purpose languages used in written translation systems. The following is an example of text input to the YACC written translation system (Johnson and Lesk 1978).

```
% token DIGIT
%%
date : number '/' number '/' number
         {date ($1, $3, $5) ;};
number : DIGIT
         {$$ = $1;};
number : number DIGIT
         {$$ = 10 * $1 + $2;};
```

The nonterminal symbols in this grammar are `date` and `number`; the terminal symbols are `DIGIT` and /. A program fragment is given at the end of each grammar rule, and these program fragments compute the meaning or value of the nonterminal symbols. The variable $$ refers to the nonterminal symbol on the left-hand side of each rule, while $1, $2, . . . , $*n* refer to the first, second, or *n*th symbols on the right-hand side of the rule, respectively.

This input text may then be processed by a YACC parser generator, which generates a program able to read dates, convert them into a suitable form, and store them in the computer, provided that the digits are first read by another program returning the value of each digit.

2.6 Two-Dimensional Positional Languages

In two-dimensional positional languages the input information corresponds to a given position in two-dimensional space. This space is displayed on a screen. The correct position of the information is generally indicated by means of a cursor controlled by a keyboard, through a joystick, or a mouse. Other methods include use of a lightpen or touch-sensitive screens.

Two-dimensional positional languages have many uses, the most important of which include systems for filling in forms, systems for screen editing, and two-dimensional query systems.

In form-filling systems the user is provided with a format map displayed on a screen and he/she can then insert the appropriate data in free areas. The format map is protected and cannot be inadvertently altered from the keyboard. After filling in the form the user presses a special key and all input data are transmitted to the computer. This type of technique is very easy to use. A typical "form" is shown below; note that the user can only put data between the square brackets [ ].

```
NAME [                          ]

FIRST NAME [                    ]

BIRTHDATE

     DAY [      ]  MONTH [          ]  YEAR [      ]

PERSONNEL CODE [                ]
```

The second type of two-dimensional positional system involves editing on a screen. A screen editing system displays part of a file on the user's screen and allows him/her to make changes at the position indicated by the cursor. There are three principal types of

command in a typical screen editing action language (Pearson 1980): cursor movement commands, text movement commands, and text modification commands. In some cases cursor movement commands can be replaced by the use of special keys on the keyboard (Altair Software Distribution Company 1977).

As our final example in this section we consider a two-dimensional query system, known as the Query-By-Example system (Zloof 1976), which is used as an interface between user and data base. In order to query the data base the user inserts a possible answer in the skeleton of the data base displayed on the screen.

As an example, the skeleton of the data base used earlier is given below.

PRESIDENTS	NAME	PARTY	HOME-STATE

Here PRESIDENTS is the name of the relation, and NAME, PARTY, and HOME-STATE are the names of the domains. To obtain information the user should fill in the skeleton using an example element (a variable), which must be underlined, and a constant element, which should not be underlined. In addition, the function "P." (print) must be inserted before the example element to indicate that this class of data forms the output.

As an example, assume that the user wishes to print out the names of the Democratic Presidents of the USA since 1956 using the relation PRESIDENTS; he/she just fills in the skeleton with P.NIXON (the name of any President would do) and DEMOCRAT.

PRESIDENTS	NAME	PARTY	HOME-STATE
	P. NIXON	DEMOCRAT	

The answer of the system should be:

NAME
KENNEDY
JOHNSON
CARTER

3 MAIN FEATURES OF DISPLAY LANGUAGES

In this section we consider the output side of the user–application software interface. The information produced by the computer to provide the user with feedback and other assistance we shall call the *display language*.

The display language must be able to perform a number of distinct functions. It should be able to:

- format the dialogue document (the printed record or screen display of the statements made by both user and computer)
- assist the user to input data and commands
- respond to the user after receiving valid input
- provide error messages
- provide "help" facilities

In this section we discuss the ways in which a display language can best fulfill these functions.

3.1 Formatting the Dialogue Document

There are a number of factors which help to produce a well-formatted dialogue document (Hebditch 1979).

1. Logical sequencing. The dialogue document contains several different types of information, and this information should be arranged in as logical a sequence as possible. One example of bad sequencing would be to blend input and output text.
2. Distinguishing input from output. It is very useful and improves legibility to distinguish inputs (action language phrases) from outputs (display language phrases). The ways in which this can be done depend on the type of terminal available. Possible methods include the use of lower-case characters for input and upper-case characters for output; underlining either input or output; using different colors or different line densities for displaying input and output, and so on.
3. Spaciousness. The whole two-dimensional space of the dialogue document can be used for output. Use of a tabular format can improve legibility; for example, if a menu is included in the output it could be presented as a table.

3.2 Assisting the User to Input Data and Commands

The assistance given to the user depends on the user–computer interface. For example, if an answer language is used as an action language, the form of the desired input is specified in the question asked by the computer. Another possibility is that the input language may contain keywords; in this case the system can be designed to assist the user through rapid keyword recognition. There are five forms of keyword recognition.

1. The whole keyword mode. In this case the user is obliged to type the whole keyword.
2. The anticipatory mode. This mode requires the user to type just enough characters for the command to be uniquely specified. The system then automatically fills in the remainder of the keyword.
3. The fixed mode. The keywords are chosen such that it is possible to recognize each keyword in a fixed number of characters.
4. The demand mode. This mode requires the entry of a special character to initiate recognition after the first part of the keyword has been typed.
5. The single-character mode. This mode allows high-speed single-character recognition of the most commonly used keywords. This mode may be used only when the keywords begin with different characters.

Another method of system assistance involves the use of *noise words*. When the system recognizes the first part of an input phrase, it can generate some words, called noise words, to tell the user what information is awaited by the system. For example, in the input command

```
CREATE LINE from x1 to x2
```

the words *from* and *to* could be generated by the system as noise words on recognizing the phrase `CREATE LINE`. The noise words prompt the user into entering data in the correct manner.

As mentioned earlier, the system may be designed to help the user by assigning default values to missing arguments, or by supplying missing information on the basis of previous commands. Whenever this happens the system should inform the user and ask him/her for confirmation.

3.3 Responding to the User after Receiving Valid Input

The system should provide regular feedback to the user on receiving valid input. The response should contain the following information:

1. Confirmation that input has been received. The system should confirm that the input is valid and has been accepted. In cases where misunderstanding is possible, as, for example, with natural language interfaces, the system can output a question and ask the user for confirmation.
2. Information about the unavailability of resources. If a process requested by a user involves the use of resources such as files or peripheral devices, the user must be informed if these resources are not available and, if necessary, *why* they are not available.
3. Output data. The output data can either be displayed on the user's terminal or by means of some other output device. In the latter case the user should be told where and how to obtain his/her results.

4. End information. When the process initiated by the user comes to an end, information about the mode of termination (normal, failure of the system, error in input, etc.) is useful.

3.4 Providing Error Messages

The system must anticipate errors in every piece of the input; sophisticated techniques must be used to handle these errors. There are three possible levels on which errors can be handled:

1. Error detection. The system must take great care to ensure that every error is detected.
2. Error recovery. After an error has been detected in the input text it is desirable to continue processing the remainder of the input without "pseudo-error" indications.
3. Error correction. Some errors may be corrected automatically, but in this case the system must ask the user for confirmation because it is possible to introduce insoluble problems through automatic error correction.

After an error has been detected the system must inform the user exactly and clearly of the nature of the error. Hebditch (1979) provides some guidelines on error-reporting techniques:

1. Avoid giving error messages in code and thus the need to refer to manuals.
2. Make error messages as self-explanatory as possible.
3. Error messages should be specified by the system designer, and the ease with which they may be understood and used checked with the potential users.
4. Errors should be detected as quickly as possible.
5. Avoid the need to rekey valid input during the error-correction process.
6. Recheck everything after correction.

3.5 Providing "Help" Facilities

Any software system must be properly documented in order to be usable (Cohen 1976). To document a large system is not an easy task, and it is made more difficult if the system is designed to be expanded by its users. Any printed documentation of such a system would be outdated before it was published and therefore the system itself must be capable of providing documentation that is guaranteed to be up to date.

In general, the user needs to know three things (Watson 1976):

- what he/she has already done
- what he/she is doing now
- what he /she can do next

The system should therefore provide information in the following three areas:

1. *Information space*. The user needs to know where he/she is in information space and which part of the information available is being displayed to him/her. The user arrived at his/her present position from a series of previous positions, and he/she may want to be able to return to these positions as well as to be able to move on. It is possible to achieve this by organizing help facilities in a tree structure. Each information node in the tree contains an explanation of a specific part of the system. The tree structure provides easy access to information about a specific topic.
2. *Subsystem or tool space*. In systems containing many tools (or subsystems), the user needs to know which tools are active, which tools he/she has used previously, and which subsystems can be entered from the present position. Each subsystem has a name and contains a number of related commands. In an ideal case all of the tools would operate on information in the same file because this would make it easier to move from one tool to another.
3. *Input syntax space*. Several ways in which the computer may help the user to formulate input have been described in a previous section. If, however, there is still some uncertainty about the basic concepts or the vocabulary, the user can employ the help facilities described above, either by specifying the concept causing difficulty or by making a more general request for help. In the latter case the system could make use of the information input up to this point to select the information required by the user.

In data-base management systems the user should be kept informed about the semantics of the data stored in the data base. In the data-base management system INGRES (Stonebraker *et al.* 1976) information about relations is available and may be used in the same way as help facilities which specify the names of the relations only.

4 CONCLUSION

Although we have discussed many of the issues concerned with user–application software interfaces, there are numerous aspects which we have not mentioned. This is largely because we have concentrated on interfaces in which alphanumeric texts are used as a means of communication. The main problem in communicating with a computer using alphanumeric texts is the great difference between the speed of the input and the speed of the output. While the output speed can be very high (thousands of characters per second), the speed of input via a keyboard is very low (less than ten characters per second). This drawback can be partly reduced by using single letters in an action language, for example, 'F' for `FIND` , 'P' for `PRINT` , and so on. However, this reduces the legibility of the dialogue document and can only be used by frequent users.

Graphics provide another promising medium for user–computer interfaces. Graphics can be used within display languages, action languages, or both. We have already discussed two-dimensional positional languages, in which simple graphics are used as part of a display language. Communication in such systems is both simpler and faster than using

alphanumeric texts, as can be seen by comparison of a line-oriented editing system with a screen-oriented editing system.

The second problem in communication between users and application software is the selection and design of an appropriate language. Computers, especially small-scale computers, are increasingly being used as everyday tools in offices, businesses, and management. Most people using these systems have little or no knowledge of data processing. It is therefore desirable to design software systems with a nonprocedural interface for these applications, and a natural language seems to be the most appropriate. However, because of the problems involved in implementing natural language interfaces even on large-scale computers, we must suppose that formal languages will remain widely used in the future. Thus it is very important to design any language to be used by nonskilled operators so that it follows the natural language as closely as possible. The interested reader may find a more extensive discussion of languages designed for use in offices in Rohlfs (1979); the design of languages to be used in managerial systems is treated in more detail in Keen and Hackathorn (1979) and Blanning (1979).

ACKNOWLEDGMENTS

I am grateful to Göran Fick for many useful and valuable suggestions during the preparation of the manuscript, to Ronald Lee and Michael Dempster for critically reviewing the manuscript, and to Miyoko Yamada for her help in preparing this report.

REFERENCES

Addis, T.R. (1977) Machine understanding of natural language. International Journal of Man–Machine Studies 9:207–222.

Altair Software Distribution Company (1977) Altair Word Processing Package. Altair Software Distribution Company, Atlanta, Georgia, USA.

Astrahan, M.M., M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, J.L. Traiger, B.W. Wade, and V. Watson (1976) System R: Relational approach to database management. ACM Transactions on Database Systems 1:97–137.

Blanning, R.W. (1979) A language for describing decision support systems. Informal Workshop on Decision Support, Department of Decision Sciences, The Wharton School, University of Pennsylvania, USA.

Bobrow, D.G. (1977) GUS, a frame-driven dialog system. Artificial Intelligence 8:155–173.

Chamberlin, D.D. (1976) Relational data-base management systems. Computing Surveys 8:43–63.

Codd, E.F. (1977) Seven steps to rendezvous with the casual user. In Data Base Management, edited by J.W. Klimbie and K.L. Koffeman, pp. 179–199. North-Holland Publishing Company, Amsterdam.

Cohen, S. (1976) Speakeasy – A window into a computer. AFIPS National Computer Conference and Exposition, Conference Proceedings 45:1039–1047.

Fick, G. (1980) The challenge of low-cost computers to the organization. In Human Choice and Computers 2, edited by A. Mowshowitz, pp. 17–19. North-Holland Publishing Company, Amsterdam.

Fitter, M. (1979) Toward more "natural" interaction systems. International Journal of Man–Machine Studies 11:339–350.

Gaines, B.R. and P.V. Facey (1976) Programming interactive dialogues. In Computing and People, proceedings of a conference held at Leicester Polytechnic. Edward Arnold, Leicester, England.

Hebditch, D.L. (1973) Terminal languages for data base access. In Data Base Management, Infotech State of the Art Report 15. Infotech International Ltd., Maidenhead, Berkshire, England.

Hebditch, D.L. (1979) Design of dialogues for interactive commercial applications. In Man/Computer Communication, Infotech State of the Art Report 2, pp. 171–192. Infotech International Ltd., Maidenhead, Berkshire, England.

Infotech International Ltd. (1979) Man/Computer Communication, Infotech State of the Art Report 1: Analysis and bibliography. Infotech International Ltd., Maidenhead, Berkshire, England.

Johnson, S.C. and M.E. Lesk (1978) Unix time-sharing system: Language development tools. Bell System Technical Journal 57:2155–2175.

Keen, P.G. and R.D. Hackathorn (1979) Decision support systems and personal computing. Working Paper 79-01-03. Department of Decision Sciences, The Wharton School, University of Pennsylvania, USA.

Kowalski, R. (1979) Algorithm = Logic + Control. Communications of the ACM 22:424–436.

Kraegeloh, K.-D. and P.C. Lockemann (1978) Access to data base systems via natural language. In Natural Language Communication with Computers, Lecture Notes in Computer Science 63, pp. 49–86. Springer Verlag, West Berlin.

Lehmann, H. (1978) Interpretation of natural language in an information system. IBM Journal of Research and Development 22:560–572.

McCracken, D.D. (1978) The changing face of applications programming. Datamation 24: 25–30.

Miller, L.A. and J.C. Thomas (1977) Behavioral issues in the use of interactive systems. International Journal of Man–Machine Studies 9:509–536.

Olle, T.W. (1973) A summary of the state of the art in data base management. In Data Base Management, Infotech State of the Art Report 15, pp. 215–223. Infotech International Ltd., Maidenhead, Berkshire, England.

Pearson, M. (1980) Using the Computer to Communicate: An Introduction to Text Processing at IIASA – The "edx" and "nroff" Programs. WP-80-111. International Institute for Applied Systems Analysis, Laxenburg, Austria.

Rohlfs, S. (1979) User interface requirements. In Convergence: Computers, Communications and Office Automation, Infotech State of the Art Report 2, pp. 165–199. Infotech International Ltd., Maidenhead, Berkshire, England.

Schneiderman, B. (1978) Improving the human factors aspect of data base interactions. ACM Transactions on Database Systems 3:417–439.

Sprague, R.H. (1981) A framework for research on decision support systems. In Decision Support Systems: Issues and Challenges, edited by G. Fick and R.H. Sprague, pp. 5–22. Volume 11 in the IIASA Proceedings Series, Pergamon Press, Oxford, England.

Stonebraker, M., E. Wong, P. Kreps, and G. Held (1976) The design and implementation of INGRES. ACM Transactions on Database Systems 1:189–222.

Teitelman, W. (1979) A display oriented programmer's assistant. International Journal of Man–Machine Studies 11:157–187.

Thompson, K. and D.M. Ritchie (1975) Unix Programmer's Manual. Bell Telephone Laboratories.

Waltz, D.L. (1978) An English language question answering system for a large relational database. Communications of the ACM 22:526–539.

Watson, R.W. (1976) User interface design issues for large interactive systems. AFIPS National Computer Conference and Exposition, Conference Proceedings 45:357–369.

Winograd, T. (1979) Beyond Programming Languages. Communications of the ACM 22:391–407.

Zloof, M.M. (1976) Query-By-Example – Operations on hierarchical data bases. AFIPS National Computer Conference and Exposition, Conference Proceedings 45:845–853.

THE AUTHOR

Bořivoj Melichar studied electrical engineering and control engineering at the Czech University of Technology and received a diploma in electrical engineering in 1964. In 1978 he completed a dissertation on the translation of cooperative lists into programming languages. Since 1964 he has been an assistant professor in computing at the Czech University of Technology in Prague. Dr. Melichar's present interests include the syntax and semantics of computer languages and the design and implementation of computer language compilers.