Guidelines for the design of interactive systems

J. Hilden

University Institute of Medical Genetics, University of Copenhagen, Raadmandsgade 71, DK-2200 Copenhagen N, Denmark

On the basis of experience with a special-purpose interactive processor the author makes several suggestions that users are likely to welcome. A central idea is that of having one mechanism for handling input, whether program, data, or error corrections, and to provide all language facilities at all input occasions. It is shown that the interactive read operation, with its prompting message, evolves naturally into a combined write-read operation. A section is devoted to a somewhat unconventional data retrieval (re-display) routine. An attempt is made to set up a conceptual framework for the various suggestions made here as well as for those of others.

(Received April 1974)

Introduction

The purpose of the present paper is to draw attention to some features that can make a conversational computer system a pleasant tool to work with, with emphasis laid on processors designed to perform tasks of some complexity, particularly programming language implementations. The GENEX processor built by the author is used as the point of departure. This conversational system allows formula manipulation in probability calculus. It has been designed to assist geneticists in deriving the type of probability formulae that the Mendelian laws and other basic laws of inheritable traits give rise to. For descriptions of the processor, seen from various points of view, the reader is referred to Hilden (1973, 1976a, 1976b).

The available input/output equipment will inevitably influence the choice of dialogue facilities. For instance, certain facilities are highly desirable in a character display terminal system and quite unnecessary where output is on paper, say in a teletype-based system. Some of the design decisions discussed reflect the specific environment in which the GENEX processor originated, viz. UNIVAC's 1108 (later 1110) computer system as seen through the UNISCOPE 100 character display unit. All data that are held on its display screen can be changed, moved around, and transmitted to the computer—one line at a time—whether they were originally written on the screen by the computer or via the user keyboard. These characteristics imply that the GENEX processor evolved in an environment with excellent local editing facilities but with no hard-copy production. Nevertheless it is probably not too difficult to modify the suggestions we are going to make to suit other physical environments. For brevity's sake, this point will not be mentioned further.

Some papers on design of interactive programming facilities will be referred to repeatedly. They all resemble the present text in that they try to show that the facilities of a particular language or language implementation will meet typical user demands. There is a high degree of agreement between the proposals, although terminology and taxonomy of goals and means vary a great deal. In the following these papers are quoted only in case of conflicting views, or to point out how a particular proposal fits into the conceptual framework of this paper, or—naturally—when a general point we want to make cannot be illustrated by examples taken from the GENEX processor. The papers are: the one by Reinfelds *et al.* (1970), who explain the philosophy of the AMTRAN language, the one by Moore and Main (1968), whose analysis leads to the TCL language proposal, and two papers discussing interactive implementations of existing 'batch' languages: Barron, 1971 (FORTRAN) and Cuff, 1972 (PL/I).

The cost of programming flexible interactive facilities appears to be small in comparison to the basic cost of a sophisticated processor. The GENEX system, which was written in FORTRAN, including a simulated virtual memory administrator, was implemented by one man (the author) over a period of some 18 months, and less than one fifth of the effort is estimated to have been spent on adding the facilities discussed in this paper to the basic language interpreter and its data-manipulating subroutines. Otherwise, it is scarcely possible to make useful statements about cost. Most of the papers cited could, as a matter of fact, be criticised for not being specific about the cost question.

The ideas with which we are concerned here are largely independent of the nature of the data being communicated—narrative, formulae, pictures, sound—so the vast literature on interactive graphical techniques as well as voice communication could be considered relevant to our discussion, but no explicit attempt has been made to cover interactive computer applications employing messages other than character strings. We also limit our attention to one-user systems; for a recent discussion of a multi-user system, see McGeachie (1973).

Although the dialogue facilities to be discussed are highly inter-woven, in use as well as in implementation, five main areas can be distinguished.

1. *The overall pattern of information exchange*

i.e. the traffic rules of the man-machine dialogue, so to speak.

2. *'Memory' facilities*

i.e. facilities that relieve the user of taking notes during the session, of retyping data, etc. If nothing could go wrong the list would end here. In practice, it is necessary to consider:

3. *how to avoid losing control,*

4. *error handling*

5. *debugging*

to the extent that this last subject is not covered by 1-4. Separate numbered sections are devoted to each of these areas. Finally one ought not to forget the area of

6. *facilities for user instruction.*

1. The overall pattern of information exchange

1.1. *GENEX message structure and general mode of operation*

According to the standard definition, a message is the smallest volume of data that can be submitted to a processor (man or machine) for processing. An input message to the GENEX processor is known as an input *segment*. Before discussing the various interactive facilities it is necessary to say a few words about this concept. A segment consists of one or more lines. At the one extreme an entire GENEX program can be placed in one large segment; at the other extreme a segment may

contain just a single data item such as 'JOHN' or '5' (there is no sharp distinction between data and program in the GENEX language). The user normally indicates the end of a segment by placing a special end-of-segment character at the beginning of its last line. This character, which does not otherwise affect the processing of the segment, will be left out in our examples. Within a segment the format is free, spaces and line boundaries being ignored.

In principle, the processor interprets one input segment at a time. After completion, it asks for another segment; let us term this an *idle* state input request. The net effect is cumulative, in the sense that the output generated in the course of a lengthy user-processor dialogue is no different from what it would have been, had the entire sequence of instructions been presented to the processor in one large segment.

This simple pattern of man-machine alternation is often broken by the processor interrogating the user in the middle of a piece of processing—a so-called *busy* state input request. For our purpose this type of segment input request is far more interesting than its idle state counterpart—as later sections will show—because the user gets a chance to interfere with a computation not yet carried to an end. The unified design to be described implies that it makes little or no difference whether the request is issued as part of a *read* operation or because the processor is in some kind of trouble.

Reinfelds *et al.* (1970) distinguish *three modes of interactive operation*: (*a*) the execute mode ('desk calculator' mode), in which instructions are executed as they are entered, (*b*) the suppressed (or delayed) mode, which allows the user to construct a program for execution at a later time, and (*c*) a special checking mode, which allows the user to execute parts of suppressed programs while they are being constructed. Moore and Main (1968) are able to fuse modes (*a*) and (*c*) by means of a clever 'statement group' device. This distinction is not applicable to GENEX, because all segments are executed immediately (mode (*a*)). In particular, a procedure declaration may be executed, emulating mode (*b*), i.e. the procedure is declared, and procedures thus constructed can be called with trial arguments, as in mode (*c*). The interactive language interpreters these authors have in mind, which are also exemplified by popular implementations of JOSS (with offspring), JOVIAL and BASIC, are characterised by the view that the typical purpose of an interactive run is to construct, check, and modify a program, until it has acquired a satisfactory form, in which it can then be saved for later use. The GENEX implementation, on the other hand, is directed towards obtaining results, rather than producing programs for future computations (although the features described in Sections 2 and 4 make program-making runs possible). In this respect it resembles APL.

Related to this dichotomy (program versus result production) are the problems of interpretation versus compilation (with intermediates), and of total versus partial recompilation after program changes. Our excuse for not discussing these problems, which Barron treats in detail, is that they do not concern the user—except indirectly via response time, as Barron points out. Compilation, however, is becoming less important these days as contemporary languages tend to become more compact (APL) or specialised (SNOBOL). At the same time efficient ways of interpreting semi-compiled code are becoming more desirable, but hardware designed for this purpose is not generally available, possibly because language-makers have not been able to state what they need.

To make GENEX easier to learn it was decided that any piece of output that could represent a (normal) computational result should agree with the rules of the input language (otherwise the geneticist would have to learn two languages). Also the fact that output can be changed at the terminal, if desired, and transmitted back to the computer spoke in favour of this decision: it is very convenient for the user to be able to hand the computer's answers back to it, knowing that they will not only be syntactically correct but also have the same meaning when re-input as they had when output Such *re-interpretable output* is common in data structure handling languages for scientific application (LISP, APL). Already the PRINT RESULTS/READ DATA statement pair of MAD, the forerunner of the NAME LIST device of FORTRAN IV, which was further developed in PL/I, represented a step in this direction. On the whole, however, re-interpretable output is an exception in numerical or commercial languages, even when they are implemented for experimental interactive programming. Future language implementers ought to consider this possibility.

Input cancellation facilities must be provided in any interactive system, in which the unit of physical input transmission is smaller than the unit of execution: as far as practicable the user should be allowed to reconsider. In GENEX, input transmission is by lines, interpretation by segments. Therefore the user should be able to revoke a segment some lines of which have already been received by the computer. To this end, a line beginning with a special character can be entered. Another type of abortive segment is the re-display request (Section 2.3). Such requests are attended to by a segment reader subroutine; like cancelled segment lines, they are not allowed to pass beyond this subroutine—which is therefore capable of independent conversation on a shallow level. Any sophisticated interactive system is likely to offer more than one 'layer' of conversational facilities.

1.2. *Busy state conversations*

The data on which a batch mode user program is going to operate must be prepared according to a fixed scheme, defined at the moment of program writing. In an interactive run, on the other hand, the information needed to solve a given problem can be read in piecemeal as the need arises, provided the system has got means for indicating which of several *read* instructions in a large program is responsible for a given input request. The processor must, so to speak, be able to ask suitable questions. (In some terminal installations an additional reason for explicit prompting is that the user has no other way of discovering that the processor has halted and would therefore, if no signal were given, begin to wonder why nothing happened). Realising that an interactive *read* operation tends to become a combined *write-read* operation, one is prepared to go one step further, attaching a full-fledged *write* operation to every *read* operation. The GENEX *read* instruction takes the form

READ: operand

and causes the operand value to be written (displayed), whereupon the processor halts, expecting a reply. A small GENEX dialogue, involving a piece of program that adds 1000 to whatever is keyed in, could look as follows, 'SHOW:' being the standard write (display) command.

input	*output*
SHOW: (1000 + READ:WHAT)	
	WHAT
444	
	1444

To achieve complete interactivity, however, not only must the program be able to say: 'In order to compute the desired results I must know . . .', but the user must also be given provisions for replying: 'Well, to answer that question of yours *I* must know . . .'. In other words, after a busy state input request he must be able to enter into a parenthetical conversation with the computer, just as he would do with a human assistant asking for his advice. In either case this conversation

necessarily consists of one or more messages (utterances) on his part, to which the assistant or computer responds in action or in words, followed by his final reply message. Each parenthetical user message must be accompanied by a suitable indication that this is not yet the final reply: GENEX employs a '> > >' replacing the usual end-of-segment mark.

In our case it is not appropriate to say that such busy state conversations are carried on in a special immediate or 'desk calculator' mode. As will become apparent in a moment, their input segments are treated exactly as other input segments are treated, except for two points: (*a*) it is possible to refer to elements of the program context in which the input request arose, (*b*) the last segment serves the additional purpose of conveying the user's answer to the request.

When the input language is interpreted (rather then compiled), there is no reason why the user should not be allowed to use all facilities of the language during the conversation. For instance, GENEX allows '111 + 333' to be entered instead of '444' in the small example above. The *busy state algorithm* which is brought into action when the operand of 'READ:' has been displayed—and, in general, whenever the GENEX processor halts expecting a reply that will allow it to proceed—can be outlined as follows:

(Step 1) Let the user key in a segment.

(Step 2) Interpret the segment according to the rules of the GENEX language. (Since the segment may itself contain a read instruction or cause an error situation, the present algorithm can be entered recursively).

(Step 3) Is '> > >' present? If so, revert to Step 1. Otherwise go to Steps 4 and 5.

(Step 4 will be explained in Section 4).

(Step 5) Take the value of the segment as the user's answer: in the case of a *read* operation the value of the segment becomes the value of the phrase 'READ: . . .'; proceed. (Like any GENEX language phrase, the contents of an entire segment has a value—as in ALGOL 68 or LISP).

The potentialities of this scheme are illustrated by means of an example, written in a dialect of ALGOL which needs no further explanation. Consider a procedure,

```
integer procedure abc (x, y, z); value x, y, z;
      integer x, y, z;
begin Boolean B;
  B := y < x;
  abc := 0;
  abc := read ('abc?');
Q : if B then global := global + 1;
end
```

Assume that this procedure has been called by

```
abc (100, 50, −2);
```

the conversation that ensues might look as follows:

```
input                          output
                               abc?
> > > show (global)
                               3417
if B    then   y else z
```

At exit from *abc*, *abc* = 50 and *global* = 3418, since *B* = **true**. An entirely different effect would be obtained if the user ventured to input something like:

```
input                          output
                               abc?
> > > global := 0; comment no output produced here;
begin B := false; goto Q end
```

Being a label-free language, GENEX does not provide the refinement exemplified by the jump out of context to label *Q*, so the above algorithm need not cater for such jumps.

Thus, in an interactive interpretative system it is natural not only to place all language facilities at the user's disposal at each input request, but also to let the ensuing conversation as well as the closing reply be interpreted in the context of the instruction that causes the input request, insofar as the meaning of variable names, etc. is concerned. In addition, the incremental nature of interactive computations makes it desirable to be able to declare new entities or to change declarations. In the case of an ALGOL-like language, the variables and labels of all currently active blocks ought to be accessible, for instance via the QUALIFY device described by Cuff, but declarations are likely to be disallowed—as in Cuff's PL/I system—due to difficulty of implementation. (With GENEX, scope problems do not arise; declarations are permitted in busy state conversations, and they are automatically global).

In compiling systems part of the facilities described here can be provided at a minimum of cost. Using the NAME LIST device one can write quite satisfactory, though rather clumsy, interactive FORTRAN programs that allow the user to examine and modify variables.

2. 'Memory' facilities

As stated in the introduction, the devices we are going to consider under this heading serve a common purpose: to enable the user to retrieve relevant data at proper time and place. He should never be forced to take extensive notes during the run—or to rely on his own memory. Nor should he be forced to type the same data twice. Three functions can be distinguished.

2.1. *Transfer of data, stored somewhere in advance, into the run*
As a minimum it should be possible for the user to get hold of procedures and data stored in his personal file.

2.2. *Saving of data generated in the course of the run*
(the converse of 2.1). Whenever a certain option is in force the GENEX processor will copy its input and output into a report file for later printing. Copying of input is essential, for otherwise the user would be unable to document the validity of his results.

2.3. *Revival of data from an earlier phase of the given run*
The remainder of the section is devoted to this topic.

Because of the small capacity of the Uniscope display (12 lines of 80 characters each) it is useful to have a facility for re-display of input and results that are no longer on the screen. Since re-displayed data can be changed, if desired, and re-transmitted to the computer, the re-display facility which we are going to discuss is really a re-use facility (see Section 1.1).

Input and output can be re-displayed, one segment at a time. By an output segment we mean one or more lines generated by one activation of a write command, typically 'SHOW: . . .'. Cancelled input segments are not available for re-display. Nor are re-displays or re-display requests. Input is displayed in compact form, longer stretches of blanks being suppressed. An option governs whether or not output will become re-displayable.

A re-display request is permitted whenever input is expected. The request is a free-format input line that takes the standard form

```
<segmentlabel
```

In many interactive text editors or compilers lines are numbered, either automatically or by the user, and most often the line number is displayed each time a line is accessed by the user. In view of the fact that the GENEX user is not interested in segment numbers until he decides that he wants a re-display, it was deemed preferable not to take up screen capacity—and

detract the user's attention—by displaying segment numbers of transferred segments. On the other hand, serial numbering was felt to be indispensable. Therefore it was decided to do it behind the scenes, as far as possible. In addition, it was recognised that the user might sometimes prefer to reference a segment by its contents or by some designation of his own invention. These considerations led to the following approach.

Segment labels are associated with such segments as must be available for re-display (see above), partly automatically, partly on user request. They are generated—and the user is informed of the automatically generated ones—according to Rules 1-4.

Rule 1
Serial numerical labels are generated automatically. Input segments are numbered from 1 upwards, output from 200 upwards. The successor of input segment 19 might for instance run:

PROCEDURE; MAX=ARG'1−ARG'2#ARG'1#ARG'2

Being the 20th input segment it can be retrieved by entering

<20

After display of the segment, the processor asks the user obligingly if he wants the next segment to be displayed also. To be specific, the message

<21?

appears on the screen, which, if re-transmitted by the user, triggers a re-display of the next segment, and so on.

Rule 2
A non-numerical label may be extracted from the *input* segment itself and recognised in later re-display requests. This is done heuristically: the processor tries to guess under what designations the user would wish to refer to the segment. In particular the initial 'word' of the segment, if any, is always retained. As a matter of fact, the procedure declaration of the above example (which declares a procedure, MAX, that finds the largest of its two arguments) can be retrieved, and the '<21?' message obtained, by any of the three requests:

<20
<PROCEDURE
<MAX

Rule 3
A non-numerical label can be associated with an *output* segment by the user by executing a special LABEL command. For instance, the instruction 'LABEL: COPENHAGEN' will output a segment that can be retrieved at a later time by means of

<COPENHAGEN

After re-display of this segment (which itself contains nothing but 'COPENH'), the usual question appears, e.g.

<224?

which in turn enables the user to unravel the output from that point onwards, even though he has not been aware of the serial numbering going on.

Rule 4
When a segment label already in use is associated with a new segment the old association is forgotten.

In deciding which segment is desired by the user the processor behaves somewhat heuristically, so non-standard requests may often have the desired effect. If a request specifies a label not in use an IGNORED message is returned.

This re-display facility has been treated in some detail because it is somewhat unconventional. It was very easy to implement and is convenient to work with. At least it avoids bothering the user with line numbers that he does not plan to use: it is a kind of subconscious facility.

3. How to avoid losing control

The first thing that comes to mind is that a HALT command must be provided. When the processor halts it must be able to indicate where and why. The user will then investigate the situation, perhaps, before entering some data that indicate that the processor may proceed. Alternatively, he may want the processor to resume operations at a different point in the program. In a label-free language like GENEX, the latter possibility is awkward and has not been implemented; instead the current computations can be abandoned by means of the special instruction

FORCE: ESCAPE

which can be executed at any time, including after a halt. It makes the processor return to the idle state, as if all requested processing had been completed. Otherwise, the facilities we wish to have at our disposal at a halt are already provided: to equip GENEX with a serviceable HALT command it sufficed to make 'HALT' synonymous with 'READ'! (In doing so, we adopted the harmless convention that the value of 'HALT: . . .' is the value of the (arbitrary) user reply taken as a signal to proceed, see the busy state input algorithm in Section 1.2). It is hard to think of reasons why HALT and READ should not always be treated as synonyms in similar interpretive systems.

It is essential in systems of this kind to be able to stop computations that threaten to consume undue amounts of computer time—whether owing to a program error or to a misjudgment. There are various ways in which this can be done. Firstly, the *actual amount of computer time* can be measured. Either a deadline can be chosen (by implementation or by the user), after which he regains control, or the user can be given access to the computer clock. In GENEX the instruction CHECK: TIME returns as its value the amount of time consumed; there is no deadline trap.

Secondly, the *number of elementary operations performed*, or statements executed, can be used as an indirect measure of resource consumption. This popular device would be easy to put into the GENEX processor but has not been.

Thirdly, an algorithm may be implemented which tries to discover when the program is *caught in a loop*.

Consider a programming language implementation in which, under normal control, instructions are executed sequentially. Let n denote the number of instructions, the last of which is a stop instruction. By monitoring jumps a simple loop check can be implemented. If the program is looping it must necessarily make an ever-increasing number of backward jumps (i.e. jumps from a high to a low address), because the program is itself finite in size. The fact that n is finite further implies that a backward jump the destination of which is higher than the destination of the most recent backward jump is not potentially dangerous, since, no matter how many such jumps take place in succession, control will move towards high addresses and will eventually reach the stop instruction (actually, after less than $n^2/2$ instruction executions). A natural loop check therefore consists of counting the number of backward jumps to destinations less than or equal to the last previous backward jump destination and give the alarm if this count exceeds a certain limit. This is, in a sense, the best possible algorithm, subject to the restriction that the algorithm can keep record of one jump destination only.

In practice, it may be easier to test all jumps, not just backward jumps. Although in some properly working programs, the count may rise faster than strictly necessary, this modification will still catch all looping programs. It was employed in the

IBM 7094 implementation (Hilden and Jensen, 1969) of L7, an L6-like list processing language with ALGOL-like syntax, conceived by Lindblad (1968). The GENEX interpreter does not use a linear instruction storage, but traverses a syntax tree. The algorithm could perhaps have been modified to cope with pointers instead of instruction addresses. A much simpler approach is possible, however.

Label-freeness implies that a GENEX program can be caught in a loop only in case of infinite recursion. Therefore its loop check mechanism is based on counting the number of procedure calls. Actually, the interpreter need only, and does only, worry about the number of calls executed since the last occasion on which the user had control (i.e. since the last segment input request). Now, to obtain at the same time loop protection *and* undisturbed operation of correct programs that happen to involve many procedure calls, the following strategy has been adopted.

If a fixed limit of one hundred procedure calls is exceeded the user regains control after an explanatory message (busy state input request). If the user's answer (defined as the value of the final reply segment, see Section 1.2) is 'GOON', computations proceed; any other answer is taken as the value of the intended 100th procedure call, which is consequently *not* executed. (In either case the loop count is reset to zero—because the user has just had control). If this is insufficient to stop wild recursion, he can make appropriate modifications during the busy state conversation. In particular, he can redeclare the offending procedure (without causing havoc). As a last resort the ESCAPE mechanism can be used. Now, the instruction CHECK:LOOP returns as its value the loop count and has the side-effect of resetting the count to zero. This clearly enables the user to program his own loop protection, if desired. In addition, CHECK:LOOP can be inserted at suitable points in procedure bodies to avoid interruption of properly working programs known to involve many calls. On the other hand, it obviously does not allow the user to disable the built-in check as such. In a sense, then, GENEX allows its user to disable the built-in check only where looping is expected, not where it is unexpected.

In the system described by Cuff it is possible to regain control, or to influence the course of computations otherwise, by pressing the *attention button*. A new PL/I ON-condition enables the user program to handle such asynchronous terminal interrupts itself. In GENEX this type of device, however useful it might be, was out of the question, as it would have involved a considerable amount of system programming. The attention button of the UNISCOPE can, however, be used to terminate the GENEX run—or to arrest temporarily the display 'handler', for instance, if output is being produced at such a rate that it disappears from the screen too fast to be read.

4. Error handling

In designing a computer system it is important to give consideration to violations of the restrictions that the processor will impose on the human. Ideally the restrictions should be an integral part of the design. They should be as few, and therefore as general, as possible. Further, the processor's error reactions should have a natural one-to-one correspondence to the rules violated. This will facilitate both documentation and use. In particular, careful thought should be given to multi-error situations: abnormal contingencies should not be allowed to interact.

In an interactive system four categories of error reactions can be distinguished: (*a*) default actions, performed without the user being notified, (*b*) remedial actions accompanied by a warning message, (*c*) errors left for the user to correct, and (*d*) catastrophic errors. Some processors provide the user with means for moving an error type from one category to another (GENEX does not). The means for correcting errors should also be conceived as an integral part of the design.

As to GENEX *warning messages*, these always indicate the remedial action taken. For instance, so-called probability 'weight expressions' are required to be 'homogeneous'. The message 'INHOMOG. WEIGHT EXPR. NULLED' informs the user not only that he has tried to construct an inhomogeneous weight expression, but also that it has been replaced by the null expression. Adding the word 'nulled' to the message is enough to indicate this fact, so why refer the user to a manual?

A GENEX *correction request* is simply a busy state input request signalled by an error message, as exemplified by the loop check mechanism discussed above. The error message is accompanied by a small quotation of the offending source program context, kept for this purpose in each node of the syntax tree. The message normally hints at the phrase to be replaced by the corrective answer. For example, '(ALFA(1) + 5)' is meaningless if ALFA has not been declared. The resulting message may look as follows:

ALFA(1 >>> ALFA IS NOT KNOWN

The user now knows that 'ALFA' is to be replaced. After investigating the case, typically by means of re-display, he simply replies 'ALPHA', if this is the intended spelling. If the cause of the trouble is that he has forgotten to declare ALFA, he enters the intended declaration and replies 'ALFA'.

The corrective answer applies to the present execution of the incorrect phrase only. If replacement for the rest of the run's lifetime is desired, the program can be repaired (patched) by means of such reply as

PERMANENT:ALPHA

The *permanent replacement* mechanism is more general, however: any source text phrase whose value is replaced by the answer to a busy state input request can itself be permanently replaced. For instance, suppose a procedure contains the instruction 'READ: AMOUNT', which permits the user to key in a different 'amount' each time it is executed. If, in a particular run, the user decides to vary this 'amount' systematically, his reply to the first 'AMOUNT' question might be

PERMANENT: $S(R)$; $P(1) + P(2)$

indicating that this first 'amount' is the value of '$P(1) + P(2)$' and all later 'amounts' are found by calling procedure S with argument R.

What PERMANENT does is to remember a pointer to the syntax tree branch representing the phrase after colon ('$S(R)$' in our example). When Step 4 in the busy state algorithm is reached, the following happens:

(Step 4) Did the reply message contain a PERMANENT instruction? If so, change the syntax tree by substituting the new pointer for the one that points to the phrase causing the input request.

A GENEX procedure may be declared at any time; this facility and the PERMANENT repair facility clearly supplement each other. By means of the editing facilities of the UNISCOPE a re-display of the previous declaration can be changed to produce the new declaration, with little risk of introducing new errors.

In a stimulating paper devoted to the human engineering aspects of interactive systems design, Hansen (1971) stressed the principle of *data structure integrity*: a trivial mistake should not be allowed to destroy a large amount of valuable data. GENEX provides so-called assumption registers which can be used to hold encoded probability distributions. The contents of an assumption register may be a very extensive data structure, which will be difficult and expensive to reconstruct. To overwrite an occupied assumption register the user must explicitly switch off a protection mechanism. This is in keeping

with Hansen's suggestions. He, however, goes one step further and provides a 'bin' into which a data structure that becomes unreferenceable is thrown, and from which it can be recovered by a special instruction. Thus he has added a new dimension to current techniques of garbage collection! This is but a special instance of what he calls *reversible action*. He notes the desirability of a general means for undoing computations, i.e. what a user might prefer to call *regret facilities*. Such facilities can possibly be borrowed from indeterministic programming languages, but they are bound to be very expensive and are likely to pervade the entire design, if they are to be truly general.

5. Debugging

Debugging is the activity of spotting and eliminating the cause of error contingencies or of incorrect results. In the interactive case a distinction could be made between debugging aimed at producing a correct program for later application, on the one hand, and correction of mistakes made in the course of instructing the processor to perform certain 'desk calculator' operations, on the other. In GENEX the repair facilities discussed above and the detective facilities described in this section are used for both purposes. The actual editing of an incorrect program must, of course, be done afterwards, on the basis of the experience gained during the GENEX run, since the GENEX repair mechanisms only serve to make a piece of program work *as if* its source text had been corrected.

Certain kinds of information which cannot be obtained by the normal output operations of the given programming language tend to be useful in elucidating an error situation. Source text quotations or references offer a typical example. Often one may want to know from where the current procedure was called. More generally, speaking in terms of the run time stack, which many language implementations employ or could employ, much of the desired information is located in the uppermost portion of the stack. Ideally, one would wish to see the uppermost elements explained in source language and source text terms. A trace of control flow is also desirable.

While the GENEX user has all normal output facilities at his disposal when trying to find a good reply to a correction request, few facilities of the present kind are provided. To supplement the source quotation, the instruction 'CURRENT: PROCEDURE' was implemented; it returns the name of the procedure being currently executed, if any. In addition, as ARG denotes its argument list, 'ARG'1' or 'ARG(1)' gives access to its first argument, etc.

Types are checked dynamically in GENEX. Actually they must be, because it is not until execution that the types of the addends in 'READ:*A* + READ:*B*' can be matched against the requirements of the '+' operator. If an illegitimate operator/operand(s) combination is encountered, the correction request will indicate the operand type(s) and the operator. For instance, the meaningless phrase '(22 = ARG)' would cause the message:

22 = ARG >>> INVALID PHRASE, NUMERAL = LIST

(The user's answer will be substituted for '22 = ARG' in this case.) To obtain not only the types but also the *values* of the operands, a special instruction, SHOW:OPERANDS, can be entered. In our example it would display '22', followed by output explaining the current argument list. Thus the instruction effectively gives access to the two top elements of the stack.

In addition to facilities for elucidating error circumstances, the detective phase of debugging employs facilities for watching the program as it works: apart from questions of rounding error as well as some special problems presented by heuristic and time-dependent programs, it is safe to say that the programmer often gets the decisive clue to the cause of program malfunction by noting the first step at which something unexpected happens. This idea underlies the many popular *trace* devices, with which the reader is undoubtedly familiar.

GENEX provides but a simple TRACE command (possibly a misnomer), equivalent to SHOW (the normal output command) except for two modifications: (*a*) it can be switched off when TRACEing is no longer needed; (*b*) output is preceded by an octal dump of the operand. This dump is of little importance at the moment, but served a useful purpose when the processor was itself being debugged. It is possible to TRACE subphrases at all levels without changing program structure, because 'TRACE:' is transparent, that is to say,

5 + ALPHA(1) and 5 + (TRACE:ALPHA)(1)

are equivalent phrases, apart from the possible output side-effect of the latter. ('SHOW:' works the same way). Procedure redeclaration is the favourite tool for inserting TRACE instructions.

The GENEX interpreter will never (touch wood!) produce an invalid result of its own accord, because it reports for correction all illegal operator/operand combinations. However, a special invalid value results when the instruction

FORCE: INVALID

is executed. *The invalid value* can be placed as an element of a list, given as an argument to a procedure, and, of course, output. But it cannot be subjected to any kind of processing. As soon as one attempts to add something to it an INVALID PHRASE message will result. This is an automatic consequence of the dynamic type check because a unique *invalid type* is associated with the invalid value. FORCE:INVALID can therefore serve as a delayed action bomb, which returns control to the user at the proper time and place. When used as 'corrective' answer it may create a second opportunity for the user to make a permanent program repair, typically involving a larger branch of the syntax tree than the twig that has caused the alarm. In terms of a run time stack (see above), it enables the user to peel off a few elements before fixing matters. Space does not permit examples.

Experience with this device suggests that an invalid value is a useful interpretive language feature, not just for the conventional purpose of flagging undefined variables (a problem, which does not arise in the GENEX language), but also as a piece of data that the programmer can introduce at suitable places by an explicit command. One would expect it to be the more useful, in comparison with tracing of access to program variables, the more functional (and hence poor in variables) the language is. When used as a tracing tool, it raises some questions, however, one of which is: Should it be necessary to hand control to the terminal each time an attempt is made to use the invalid value? Would it not be more informative to let the invalid value propagate, a message to the effect being issued, e.g. until a conditional jump governed by an invalid Boolean is encountered?

Conclusion

Hopefully the ideas expounded in this paper will stimulate designers of conversational computer systems. The six areas listed in the Introduction and the subareas mentioned in the individual sections provide the beginning of a framework which may prove useful in itself. To those readers who have personal experience with designing such systems the material of Section 1.2 will probably appear novel. This includes the combined write-read nature of the GENEX read command, the flexible dialogue organisation laid down in the busy state input algorithm, and the emphasis on putting program entry, data entry, and error corrections all on the same footing. A user-processor interphase designed along these lines should be tidy and, in addition, easy to implement. Readers with a batch processing background may have been struck by the fact that programming

for conversational operation is more profoundly different from batch programming than one would think in advance. Consequently it is worth while to study such papers as the ones referred to here before embarking upon a non-trivial interactive system, be it administrative, scientific, medical, or general-purpose.

Acknowledgements
The staff of RECKU (the Regional Educational Computing Centre at the University of Copenhagen) have been immensely helpful throughout the years. The author is also grateful to Peter Naur and Peter Gardner for critical reading of the manuscript.

References

BARRON, D. W. (1971). Approaches to conversational FORTRAN, *The Computer Journal*, Vol. 14, No. 2, pp. 123-127. CR 22307.
CUFF, R. N. (1972). A conversational compiler for full PL/I, *The Computer Journal*, Vol. 15, No. 2, pp. 99-104.
HANSEN, W. J. (1971). User engineering principles for interactive systems, *AFIPS conference proceedings*, (1971 FJCC), Vol. 39, pp. 523-532.
HILDEN, J., and JENSEN, P. (1969). Programming i L7, *Northern Europe University Computing Center technical report C.* 535/69, Lyngby (Denmark).
HILDEN, J. (1973, being continuously updated). The GENEX Uniscope System. Xerox copies available from the author.
HILDEN, J. (1976a). Probability problem solving by computer—the GENEX system, to be published.
HILDEN, J. (1976b). Finite categorial probability problems and the GENEX algebra, to be published.
LINDBLAD ANDERSEN, P. (1968). L7—et listeorienteret programmeringssprog, *Nord DATA* 68, (conference proceedings), Helsinki.
MCGEACHIE, J. S. (1973). Multiple terminals under user program control in a time-sharing environment, *CACM*, Vol. 16, pp. 587-590.
MOORE, R. K., and MAIN, W. (1968). Interactive languages: design criteria and a proposal, *AFIPS conference proceedings* (1968 FJCC), Vol. 33 (part 1), pp. 193-200. CR 17069.
REINFELDS, J., ESKELSON, N., KOPETZ, H., and KRATKY, G. (1970). AMTRAN—an interactive computing system, *AFIPS conference proceedings* (1970 SJCC), Vol. 36, pp. 537-541. CR 20658.

Book reviews

Lang-Pak—An Interactive Language Design System, by L. E. Heindel and J. T. Roberto; 1975, 184 pages. (*American Elsevier Publishing Company Inc*, US $9.75)

Overall this is a very well presented and interesting book. However, the title is slightly ambiguous; the book in the main emphasises language design which is interactive, rather than a design system for interactive languages. Although little emphasis is placed on the latter, the system presented does not preclude their design and implementation.

The book is divided into four parts: an introduction to languages, grammars and parsing, design of languages using Lang-Pak, implementation of Lang-Pak and sizeable appendices including Lang-Pak listings in PL/I and FORTRAN. It appears to be largely aimed at the reader who wishes to include a syntax analyser in his work with the minimum of fuss and previous experience of language design. As such it is excellent and makes no attempt at taking its study of grammars any further than needed by a Lang-Pak implementor. The references included, however, give a good cross section of further reading on such topics.

For the user of a system such as Lang-Pak, the introduction to languages is both clear and concise. Terms are introduced in a simple way with plenty of examples, almost certainly leaving the novice with a good understanding of the topic.

Like many other introductions to grammars, semantics are superficially dealt with in the introduction. However, the balance is certainly redressed in the part dealing with use of Lang-Pak. This deals with the interfacing of semantic routines to the syntax analyser and the detailed design of a language. Whilst the language design sessions included are instructive, by the end of the section they pall and one has the feeling that some of the material could have been presented more concisely.

Further depth is included, specifically aimed at the Lang-Pak implementor. This is very detailed material about internal data structures and subroutine interfaces and obviously has to be studied in conjunction with the PL/I and FORTRAN listings of Lang-Pak provided in the appendices.

The book is recommended as a reference and implementation manual for those requiring a flexible, well designed language design system.

S. R. WILBUR (London)

The Design of an Optimizing Compiler, by W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs and C. M. Geschke, 1975; 165 pages. (*American Elsevier*, £6·30)

This book describes the overall structure of an optimising compiler to translate Bliss/11 source code into PDP-11 object code. In the preface the authors state that they have attempted to walk the fine line between a purely formal, implementation-independent treatment and tedious implementation detail—in this they have succeeded admirably. The book is terse, to the point, eminently readable and leaves one with a good picture of a complete system—indeed itching to go away and implement a similar system.

However the book is not for novices. Before reading it one should be familiar with the general structure of a compiler and with compiling techniques; such terms as lexical analysis, tree representation, hash tables and binding should be familiar. Short primers describing BLISS and the PDP-11 are included and give sufficient information to enable one to follow the overall design.

This book is highly recommended as a case analysis of the overall design of complete system combining known algorithms with necessary heuristics and taking into account the various interactions which arise. If you have the necessary general background and want to see a complete optimising compiler, read the book. If you are looking for a book for students' use in an in-depth study into a particular compiler then this book is worth looking into.

D. C. COOPER (London)