MAN-COMPUTER DIALOGUES FOR MANY LEVELS OF COMPETENCE

P.A.V. Hall

Information Systems Division

SCICON Ltd.

Sanderson House

49 Berners Street

London, W.1.

ABSTRACT

Man-computer dialogues are viewed as languages and the relationship between programming languages and menu plus form dialogues is shown. Syntax notation is used for dialogue description and design. It is shown how to construct a man-machine system where the user can switch freely between programming language style of interface and a dialogue of menus and forms.

1 INTRODUCTION

Interactive systems are now very common (eg. 9,10,12,13,15). The systems vary in degree of sophistication they demand of the user, ranging from programming languages like BASIC or APL to systems based on menus, forms, etc. to systems based on natural language. The various alternative forms of man-computer dialogue have been surveyed in the very useful book by Martin (10).

In many systems there is a need to accommodate both experienced users, who require a very succinct language to enable them to work fast and without frustration, and naive users who need a lot of assistance. It is desirable for the naive user to progress to advanced status without having to learn a new advanced dialogue separate from their beginner's dialogue. HELP commands, abbreviations for English-like keywords, and MACROS (eg. 10,13) are widely used: these ideas are useful, and they do accommodate a range of competences, but can we not go further ?

I found myself involved in discussions around the design of an interactive system, facing advocates of a programming language approach and advocates of a menu plus form approach. It then occurred to me that these approaches are very closely related, and it should be possible to enable the user to freely switch from menus or forms to programming language within a single system. Thus I set out to formulate my proposals in detail: this paper is the result.

For the purposes of exposition, I have invented a library retrieval problem: this example is typical of many interactive systems concerned with on-line retrieval of information (eg. 10). Section 2 introduces the example, and later sections draw upon this example.

The first consequence of looking at dialogues as languages is that all the computing industry's experience of languages and compilers can be applied. In particular, the language can be described by a syntax, and this is what has been done in Section 2.

A BNF notation has been used in Section 2, but diagrams similar to state-transition networks (14) are used later, and are important. The interaction can be designed initially as a programming language, and from this the dialogues derived: this is done in Section 2 for the example, and general rules are described in Section 3. The syntax description includes both man and computer generated symbols - the design of the dialogue is essentially a matter of deciding how much the man contributes and how much the computer contributes. Only the combination of both computer and human contributions has meaning, spelling out the intentions of the man and the required actions of the computer.

A naive user is more at home with a dialogue of menus and forms, but as he gains competence he would like to short-cut the verbosity and slowness associated with the dialogue and key in directly his instructions. This shift of behaviour is readily accommodated by viewing the dialogue as a language. The basic ideas are given in Section 4, and various software issues are discussed in Section 5 and 6.

2 EXAMPLE - Library Administration

The books held in the library are recorded in the table BOOKS, with each book described by AUTHOR, TITLE, PUBLISHER, YEAR of publication and library ACCESSION - NUMBER. People who are entitled to use the library are recorded in table SUBSCRIBERS with their NAME and ADDRESS, while actual loans are recorded in the LOANS table. Thus there are three tables with columns as below:-

BOOKS (AUTHOR, TITLE, PUBLISHER, YEAR, ACCESSION-NUMBER)
SUBSCRIBERS (NAME, ADDRESS)
LOANS (NAME, ACCESSION-NUMBER)

Note that this can be thought of as relational system in as much as its information is tabular (4,11) but as will be seen below only limited operations will be allowed.

In use, the three tables can be independently updated or interrogated to answer such questions as "does the library have any books by SMITH?" or even "who has on loan that book by SMITH?". To this end the simple query/update language of Figure 1 is designed, as a conventional programming language.

Clearly we could expect the user to type commands into the system according to the syntax of Figure 1. Instead we could design a menu/form driven dialogue offering the same facilities. The menus and forms are shown in Figures 2 and 3. These are derived directly from the syntax of Figure 1. Each menu or form is displayed with a heading showing the complete dialogue so far, so that, for example, having made the following choices, menu 1: UPDATE, menu 2: DELETE, menu 3: BOOKS, when form 1 is displayed, it will have a heading UPDATE DELETE BOOKS.

3 DESIGN OF DIALOGUES FROM A PROGRAMMING LANGUAGE

In the preceding example, we saw how an interactive system was specified by initially

FIGURE 1 Syntax and Semantics of the library administration update/query language

```
<command> ::= UPDATE <update> /QUERY <query>
<update> ::= DELETE <deletion> /INSERT <insertion>
<deletion> ::= <table - descriptor>
<insertion> ::= <table - descriptor>
<table - descriptor> ::= BOOKS <books - descriptor>
                         /SUBSCRIBERS <subscribers-descriptor>
                         /LOANS <loans-descriptor>
<books-descriptor> ::= AUTHOR ( <string> /?) TITLE
        ( <string> /?) PUBLISHER ( <string> /?) YEAR
          ( <number> /?) ACCESSION-NUMBER ( <number> /?)
<subscribers-descriptor> ::= NAME ( <string> /?) ADDRESS
                                  ( <string> /?)
<loans-descriptor> ::= NAME ( <string> /?)
             ACCESSION-NUMBER ( <number> /?)
<string> ::= <character> *
<character> ::= <digit> /A/B/.../Z/./-/,/space
<digit> ::= 0/1/2/3/4/5/6/7/8/9
<number> ::= <digit> <digit> *
<query> ::= <table-descriptor>
```

In the syntax notation, * means arbitrarily many repetitions of the construct.

SEMANTICS OF UPDATE DELETE

The table selected is searched for the entries described; all entries matching on the field values supplied, with a '?' matching anything, are found; all these entries are deleted from the table.

Semantics of UPDATE INSERT

The table selected has added to it the new entry specified; where the field value is not known, a '?' can be entered - this will have the property that on later searches, a match will always be obtained on this field.

Semantics of QUERY

The table selected is searched for the entries described, as for UPDATE DELETE: the entries found are displayed for the user in tabular form, with a suitable mechanism for handling tables too large for a single screen.

specifying the user interface using conventional syntax methods with semantic annotation, with the interactive dialogues being derived from the syntax. In this section, we abstract general rules for the design of man-computer dialogues from a programming language. The details of the dialogue will necessarily depend upon the device through which the interaction takes place, and as in all design situations, these constraints will influence the early stages of the design process and the design will be iterative.

In the design of conventional programming languages, one usually distinguishes two phases. In the first phase, basic functional capabilities are decided; what information structures are to be manipulated and what manipulations are to be permitted. In this first phase, all the essential information that the user must supply (eg. numerical values) will be identified for each command, but no consideration of how the user will supply this information is undertaken. In the second phase, the details of how the facilities are presented to the user are considered: are values identified explicitly by a label which describes them or implicitly by position, and so on. User convenience and the capability of the computer to analyse the language are both considered at this second stage.

Figure 2 Menus for Library Administration update/query dialogue

MENU	OPTIONS	NEXT MENU/FORM
1	UPDATE	Menu 2
	QUERY	Menu 3
2	DELETE	Menu 3
	INSERT	Menu 3
3	BOOKS	Form 1
	SUBSCRIBERS	Form 2
	LOANS	Form 3

Figure 3 Forms for Library Administration update/query dialogue

FORM	FIELD TITLE	INITIAL DISPLAY	ENTRY
1	AUTHOR	?	string
	TITLE	?	string
	PUBLISHER	?	string
	YEAR	?	string
	ACCESSION-NUMBER	?	number
2	NAME	?	string
	ADDRESS	?	string
3	NAME	?	string
	ACCESSION-NUMBER	?	number

In systems design, the first phase corresponds to functional specification, while the second phase corresponds to user interface description: In formal approaches to language design (eg. 1), these two phases are known as "abstract syntax" and "concrete syntax" respectively.

In principle, in designing a dialogue system, we should progress our language design as far as functional specification only, and use that as the basis for the design of the dialogues. However, it is useful to design the language with a concrete syntax as if it were a conventional programming language since, as we will see in the next section, we will want experienced users to take short cuts by enabling them to revert to the related programming language.

Thus we take as our starting point a language which has been fully specified in its concrete syntax using normal language specification conventions (eg. 6).

Four basic rules suffice in guiding our design of the dialogue.

RULE 1 - Syntax of form <class 0> ::= <class 1> / <class 2> /.../ <class n>
Make a menu one option per class on the right hand side. The syntax class names may themselves not be adequate to guide the user, and extra guidance may be necessary. This guidance could take the form of short descriptions of the classes, possibly only displayed on demand, but the guidance could employ language terminal symbols like "DELETE" etc. which are meaningful to the user. The case

```
 <class 0> ::= SYMBOL-1 <class 1> / SYMBOL-2 <class 2>.../SYMBOL-n <class n>
```

is especially useful since the terminal symbols SYMBOLi can be used to denote the options, as was done in the menus of our Example of section 2.

In other cases, terminal symbols from further down the production sequence could be used, as in

```
 <arith - exp> ::= <add - exp> / <sub - exp> / <arith - exp> /
                   / <div - exp> / <number> / ( <arith - exp> )
 <add - exp> ::= <arith - exp> + <arith - exp>
 <sub - exp> ::= <arith - exp> - <arith - exp>
 <mult - exp> ::= <arith - exp> X <arith - exp>
 <div - exp> ::= <arith - exp> ÷ <arith - exp>
```

when the options list for the <arith - exp> menu could be:
+,-,X,÷,NUMBER, (   )
NOTE: in doing this, we are really digging behind the concrete syntax and are looking at the abstract syntax and from this we are creating an alternative concrete syntax equivalent to the first.

RULE 2 - Syntax of form:= <class 0> ::= <class 1> <class 2> ... <class n>
No explicit menu for <class 0> is necessary, but the interactions for <class 1> , <class 2> ... have to be successively worked through to gather the user requirements for <class 0> .

In our example of section 2, there is no example of this rule. However, it could have been applied to all those productions which led to a form. For example, the syntax production for books-descriptor could have been rewritten as

```
 <books-descriptor> ::= <author> <title> <publisher>
                        <year> <accession-number>
 <author> ::= AUTHOR ( <string> /?) ... etc ...
```

and the production for <books-descriptor> demands Rule 2 - to obtain the information for a <books-descriptor> , each of <author> , <title> and so on, have to be acquired in turn.

These are the basic rules for a dialogue, and a dialogue could be constructed entirely from these. However, in many cases,the user can input a collection at one go - this leads to prompts and forms as in Rule 3. It is also important to keep the user aware of his progress - hence Rule 4.

RULE 3 - It often happens that the possibilities for completing a syntax class, such as <number> , are well known and that this can be input in one go. Thus for

```
 <class> ::= <number>
```

a suitable prompt should enable the user to input a number without further menus. For more complex primitives like dates, some rules of formation and a few examples may be necessary to guide the user. Even syntax classes like arithmetic expressions could be handled in this way.

Where the syntax class demands a set of such responses, we find the requirement for a form: each separate response needs to be separately prompted in a way that is meaningful to the user and as in Rule 1, terminal symbols taken from the syntax would be especially useful. The example of section 2 has three such forms.

RULE 4 - at all intermediate positions keep the user notified of the story so far. The easiest way to do this is to maintain on the display the command that he has built up so far. In a complex interaction sequence, it is very easy to get lost.

In this section, we have progressed from a programming language to a dialogue of menus and forms and similar. This process can be carried through in the reverse direction, and given a dialogue of menus, forms, etc., a programming language can be very easily derived. As we shall see in the next section, we will require both language and dialogue, and the closer these are to each other the better. Ideally, they should be designed together, rather than one first and the other afterwards. Note that Black (2) comes very close to doing what we have done in this section, but then drifts away from the essential linguistic nature of a dialogue.

4 ENABLING USER SHORT CUTS

A completely interactive system can be very tedious to use. For a new user of a system, the menus and forms and other aids are very useful, and little prior learning is necessary in order to be able to effectively use the system. However, once the user obtains knowledge and experience, it can be very irksome to have to wait while a screen full of information is displayed, to have to exercise an option by positioning a cursor, and so on. A command language becomes attractive - the user can pace the system, providing that he does not have to type excessive language keywords, and he will be able to work as fast as his typing skills permit.

In section 3, we partially recognised this requirement by Rule 3, allowing complex information groups which would be readily comprehended by the user (eg. numbers) to be keyed directly into the computer. This capability can be fully generalised to allow the user complete freedom to switch to the command language and back to dialogues as required. To see how this can be done, let us return to our example.

Example revisited

The language and dialogue of the example of section 2 can be represented by a state diagram. One form of this is shown in Figure 4. In drawing this diagram, we have had to slightly modify the syntax to break-up the productions for the descriptions. Starting at state one, the first symbol of the dialogue selects a state-transition to move to the next state, and successive symbols in the dialogue sequence cause successive state transitions. The dialogue sequence continues until State 28, the terminal state has been reached. Figure 5 relates the states to menus and forms.

In the diagram of Figure 4, four sub-diagrams have been used - these form "sub-routines", but are not essential, since the sub-diagram could have been repeated at each place it is used, to give a single state diagram.

In general, from any state in the state diagram, a sequence of symbols will "drive" the system to a new state. Thus from state 2, the sequence INSERT LOANS NAME 'JONES' drives the system to state 25.

Consequently, all we have to do to allow the user to make shortcuts and bypass the dialogues is to allow him to key in sequences of symbols at any point and allow a suitable menu or form or other prompt at the end.

FIGURE 4. State diagram for the language and dialogue of Section 2.

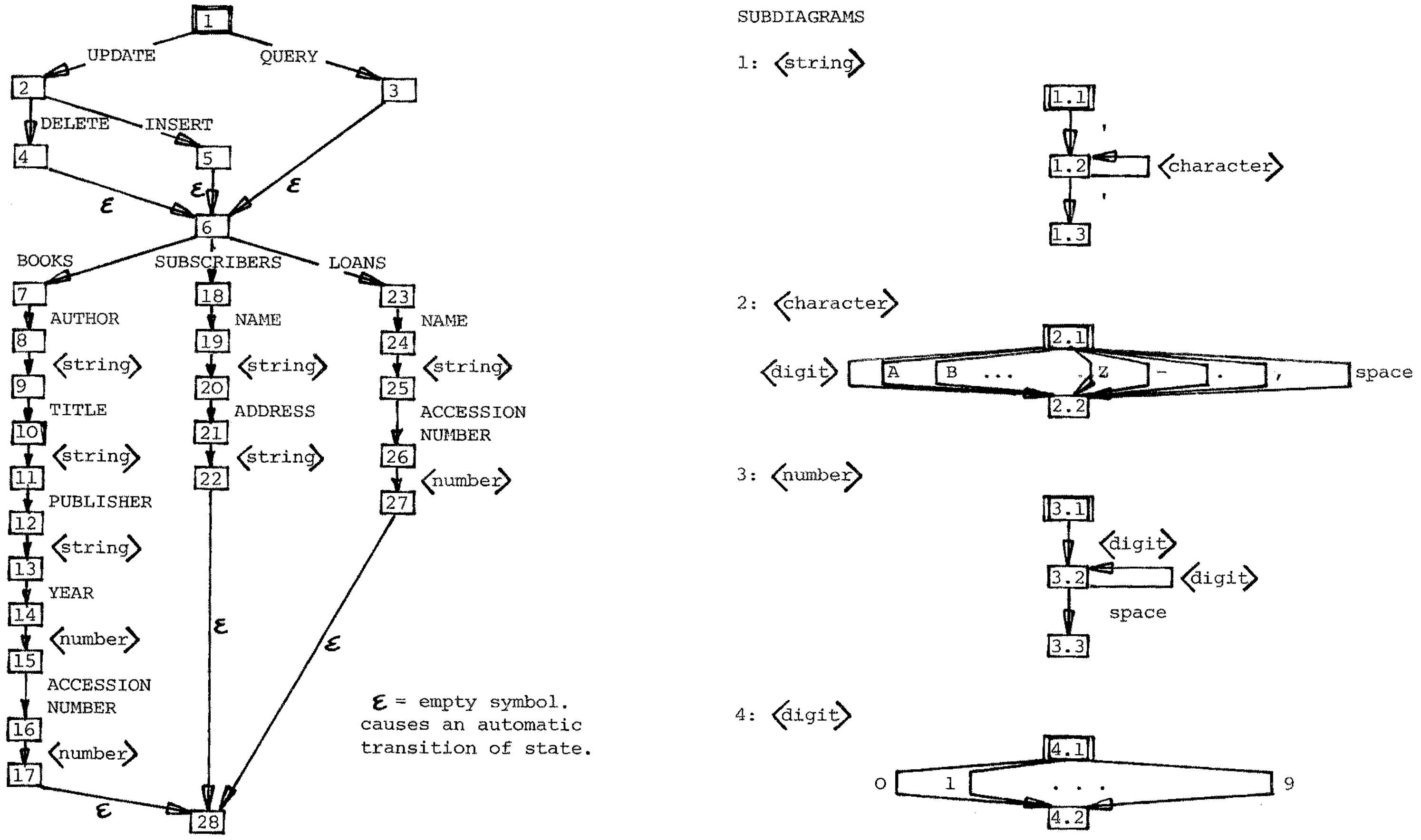

Figure 5 Correspondence between states of Figure 2, and Menus and forms of Tables 1 & 2

MENU/FORM	STATES
MENU 1	1
MENU 2	2
MENU 3	6
FORM 1	7 to 17
FORM 2	8 to 22
FORM 3	23 to 27

Hence a possible scenario for the user of our system is as in Figure 6.

It is very easy to arrange that continuations such as those of Figure 6 can be made. Details will depend upon the particular terminal. There is a cost in software, for we have to be able to parse strings now.

In general, to allow user shortcuts, we must build a generator/parser embodying the notion of state. In any particular state, a menu or form is displayed to prompt the user for the next symbol to be generated, the next state transition to be made. Optionally, the user can input a sequence of symbols to skip a few menus or forms, driving the system through several states as the symbol sequence is parsed; in the state finally arrived at, the next menu or form is displayed for the user to select the next symbol to be generated.

Note that the idea here of typing in the next few symbols is very similar to the idea of keying ahead which is possible in some systems (eg. IBM VM/CMS) though these systems still blindly go through the prompts and menus even though these should not be necessary. A limited form of command/menu alternatives is also available in some systems. (eg. DEC's PDP-11 IAS).

This idea of switching freely between programming language and dialogue can be combined with the established ideas of abbreviations, MACROs and HELP commands to provide a very adaptable dialogue. MACROs and abbreviations enable further speed-up at the programming language end, while HELP provides extra assistance at the menu and form end. We then see two modes of use of the interaction system as follows:

Naive user

Basic mode - dialogue of menus and forms. If get stuck - request HELP to obtain further information about response expected. If become familiar with part of dialogue - anticipate the following questions and revert to related programming language. Use abbreviations and MACROs if these are known.

Figure 6 Possible Scenario for Use of the system with user short cuts

STEP	USER ACTION	SYSTEM RESPONSE
1	signs on	Menu 1, in state 1
2	Selects UPDATE	Menu 2, in state 2
3	Selects INSERT and continues with LOANS NAME 'JONES'	Form 3 partially completed with NAME 'JONES', in state 25
4	Completes form with ACCESSION-NUMBER 103X	Detects error, in state 26 and recovers to prompt user with ? or number
5	Selects number	Menu 0,1,2,3,4,5, 6,7,8,9, in state 26/3.1/4.1
6	Selects 1	Menu 0,1,2,3,4,5, 6,7,8,9,Sp in state 26/3.2/4.1
7	Selects 0, etc.	cycles in state 26/3.2/4.1 until space input
8	Selects space to complete form and dialogue	In State 28 system makes INSERTion requested, and returns to state 1 for a new dialogue

Experienced user

Basic mode - programming language. If get stuck, with partially completed command or statement - system automatically throws up a menu or form to assist continuation. As gain experience - use abbreviations and other short forms, and where appropriate, define MACROs to further reduce keying effort, and thus speed-up communication.

Note that these "modes" are modes of user behaviour and not modes of the computer system. MACROs could be invoked directly from the dialogue, and requests for HELP could be made as part of a programming language style command.

Finally, a formal remark about the scope of this technique. Figure 4 is a simple finite state diagram for a finite state machine. We could draw this diagram because the language concerned was regular (6,8). For context-free languages, such diagrams become more complex, either becoming AND/OR graphs (7), or more usefully becoming state transition networks (14) which are similar to our Figure 4 but allow recursive sub-diagrams. Thus with context-free languages the notion of state becomes more complex. This does not set a limit on the computer's capabilities, but could set a limit on the user. During a dialogue, a user has to keep track of where his dialogue is taking him. With complex languages, especially of a context-free nature, we may be leading him into trouble.

5 ERROR RECOVERY

At any time, the system may discover an error, or the user on inspecting his dialogue so far may discover he has made a mistake. In both cases, there is a need to be able to backtrack to an earlier point in the dialogue to recover from the error.

The very simplest form of recovery available to the user is the editing of a response prior to its"transmission" to the computer. This editing could be local in the hardware of the terminal, or it could involve software; transmission could be simply the transmission of information from terminal to computer, or it could be by software directing the message to the segment of software that decodes commands. The act of transmission is a commitment by the user that the message transmitted is intended, and recovery after this is more complex. The user requires an ESCAPE or CANCEL key, to stop the processing he has initiated. The CANCEL could CANCEL the complete command, returning the user to the base state of waiting-for-a-new-command, or it could cancel just the last fragment of the command, with the user having to CANCEL many times if he wishes to revert to base state. A nice feature here could be the ability to edit the command or command-fragment just cancelled, and re-transmit this: this requires that the computer recovers the command and sends it to the user who then edits it using the same mechanisms as if it were an original command.

For the computer, the backtracking requirement is much simpler than the error-recovery requirement for batch compilation (6 ch 15) for it is not necessary to be able to recover and continue syntax checking as in batch compilation.

Error recovery can be expensive in implementation (see Section 6), so one has to seriously consider not supplying this - if the user makes mistakes, then he must live with them, paying a penalty in delay or the need to undo the damage caused (eg. incorrect update).

6 IMPLEMENTATION CONSIDERATIONS

The software to control a dialogue of menus and forms is closely related to the software for compilation/interpretation. Both kinds of software must have the language syntax incorporated either in the code or in a table.

In a compiler, the syntax is used for parsing - the user supplies a complete sequence of symbols, a "statement", which the software analyses using its syntax in order to understand it: internally a parse tree or equivalent will be formed. In a dialogue system the software uses the syntax generatively; to build up a statement, each time in the syntax where a choice is possible, this choice is given to the user (Section 3, Rule 1 and Rule 3). As the sequence of menus and forms are worked through, the system builds up a complete statement of what the users requirements are, performing actions on the users behalf as sufficient information is obtained. In the dialogue system, parsing as such is not performed, but the progress of the dialogue must again be recorded as a parse tree or equivalent.

It is then relatively easy to combine software for dialogue control and compilation/interpretation. A single syntax table would be used, and this table would include all syntax for the programming language together with all abbreviations and short forms, and would include all menus and forms as appropriate togèther with any HELP text. This syntax description would not necessarily be much larger than that required separately for dialogue control and compilation, since the menus and forms could be directly dèrived from the language syntax, and HELP text could be shared so as to satisfy several needs.

Error recovery requires a lot of software support. The easy part is the backtracking through the states to an earlier position in the dialogue: this requires only a suitable representation of the syntax, and the actual command being recovered from. The difficult part comes in stopping any actions being undertaken by the computer (eg. lengthy calculations or listings), and in undoing previous actions. Cancellation of actions requires that the system is left in a tidy state, as if the action had never happened. The most difficult part here is the undoing of "updates": a solution here resides in the use of "spheres of recovery" (3,5) or equivalent. However, one has to accept that recovery arbitrarily far back in time will not be possible because of storage limitations and because some updates will already have been picked up and used by other users.

From the preceding discussion, we see that the implementation of an interactive system allowing free movement between programming language and a dialogue of menus and forms is more complex than the adoption of one single mode of interaction, but it is not as expensive as implementing the two modes as separate and independent modes of interaction.

It is worth commenting on Black's system (2). While this does not give the user the flexibility we have been striving for, the software does contain interesting features. During the dialogue, a "statement" of what the user has done is built up and can be edited; a command is code-generated for some other processor to execute.

7 CONCLUSIONS

This paper has shown that programming languages and man-computer dialogues are closely related. Simple rules allow a programming language to be converted into a dialogue of menus and forms, and vice versa. Section 2 showed a simple example of an interactive system with both styles of interface, and Section 3 showed rules of construction from one to the other. Given such rules of conversion, conventional syntactic descriptions can be used for dialogue.

Having shown the relationship between programming language and dialogue, the next step in sections 4, 5 and 6 was to show how, within a single system, the user could shift freely from programming language use of the system to a dialogue of menus and forms, and back again. Traditional aids such as abbreviations, HELP, MACROs could be included to give a very flexible system capable of accommodating many levels of competence in their users. This facility has a cost in software which is comparatively small - the question is, are the benefits worth the cost involved ? For specialised systems like airline reservations, the answer is surely no, but for systems like management information systems where the user population may be very varied, the answer must surely be yes.

8 REFERENCES

1. Becik H., Bjorner D., Henhapel W., Jones C., and Lucas P. "A Formal Definition of a PL/I subset" IBM Vienna Labs, report TR 25.139.
2. Black J.L. "A general purpose dialogue processor" National Computer Conference, 1977. pp 397-408.
3. Bjork L.A. "Recovery Scenario for a DB/DC System" ACM 1973 Proceedings Vol. 28, pp 142-147.
4. Date C.J. "An Introduction to Database Systems" Addison - Wesley 1975.
5. Davies C.T. "Recovery Semantics for a DB/DC System" ACM 1973 Proceedings Vol 28, pp 136-141.
6. Gries P. "Compiler Construction for Digital Computers" Wiley, N.Y. 1971.
7. Hall P.A.V. "Equivalence between AND/OR Graphs and Context—Free Grammars" Comm. A.C.M. July 1973, pp 444-445.
8. Hopcroft J. and Ullman J. "Formal Languages and Their Relation to Automata" Addison-Wesley, 1969.
9. Infotech "Interactive Computing" State of the Art Report 10, Infotech 1972.
10. Martin J. "Design of Man-Computer Dialogues" Prentice Hall 1973.
11. Martin J. "Computer Database Organisation" Prentice Hall, 1975.
12. Pritchard J.A.T. "Selection and Use of Terminals in On-Line Systems" National Computing Centre 1974.
13. Watson R.W. "User Interface Design Issues for a Large Interactive System" Proceedings of National Computer Conference 1976, pp 357-364.
14. Wood W.A. "Transition Network Grammars for Natural Language" Communications of A.C.M. Vol. 13, No. 10, 1970, pp.591-606.
15. Zloof M.N. "Query be Example" Proceedings of National Computer Conference 1975, pp 431-438.