INTERACTION, INTERFACES AND DESIGN

by

John S. Gero[1], Warren G. Julian[2], and W. Neville Holmes[3]

[1]Harkness Research Fellow
Department of Architecture
University of California, Berkeley

on leave from:

Department of Architectural Science
University of Sydney, Australia

[2]Department of Architectural Science
University of Sydney, Australia

[3]IBM - Systems Development Institute
Canberra, Australia

1.0 INTRODUCTION

Many planning problems cannot be formulated in such a manner that a well developed optimizing technique can be applied to them to arrive at a satisfactory solution. This may be because of the structure of the problem itself, it may be because objectives and even parameters cannot be adequately described or it may be due to the need to include subjective factors. Additionally there is a class of problems for which the cost of the computation required to reach an optimum is not warranted by the expected gains.

The planning of buildings is a problem which falls into each of these categories depending on how it is formulated. For this and many other design problems we use heuristic methods.

However, whilst a heuristic can be used to commence the solution of a problem it is rare that such a heuristic can "learn" from the results of its applications. The heuristic which allows not only the inclusion of previous solutions in its experience but also allows subjective evaluations of solutions, in addition to objective numerical evaluations, is to include the user of the problem solution in the solution process. Today, this can be readily achieved through the use of interactive computing, via a time-shared computer.

When we examine the tools available to a problem solver who wishes to use interactive computing, we find that the languages currently used to achieve interaction are both clumsy and restrictive.

2.0 INTERACTION

Most interactive systems are unsuited to this type of problem solving situation because they suffer from a rigidity of system-user interface which makes it so difficult for the user to manipulate the system that instead of learning from the solutions he has been producing, he spends much of his time trying to remember input procedures - woe betide the user who loses his place in an input format. If we intend to use interaction we must provide a much better interface than is currently available via the input/output routines of the commerically offered

interactive languages.

Some of the requirements for interactive design programs have been stated (1):

"One of the prime requisites of any design program is...a facility that enables the designer to change one or more parameters independently and obtain a new set of results incorporating these parameters.

"A second requirement for such programs is that as far as possible they should be explanatory. A designer is prepared to read a limited amount of instruction but in the middle of a design problem he does not want to devote much time to learning how to use the program. Against this requirement, however, must be set the knowledge that the man who is familiar with a program will become impatient if he has to read the operating instructions every time he uses the program, and so, if possible he must be given the option of suppressing this information.

"A third requirement for such programs is that wherever possible input should be free format. Many designers have a resistance to computers and this is further aggravated by having to pay attention to decimal points, leading or trailing zeros, and counting columns when dealing with data input."

A number of attempts have been made to achieve these general requirements by writing FORTRAN routines. These have been clumsy and it would appear that an interpretive compiler such as APL (A Programming Language) (2) allows for the achievement of these through the use of a small number of user generated, general purpose functions.

2.1 Aims and Objectives in Interaction

In order to facilitate interaction between the system and the user a number of objectives may be stated:

1. the system should not fail irrespective what the user did to it;
2. the user should be able to use everyday language when communicating generally with the system;
3. the user should be able to use the technical language of the problem being solved;
4. the system should be such that the user need have had no experience or detailed knowledge of it before he uses it;
5. the system should respond according to the expertise of the user;
6. the system should provide "help" as needed;
7. the system's responses should be variable;
8. the system should accept literal descriptions as well as mixed literal and numerical input.

These objectives contribute to the general aim that the interface between the user and the system be "low profile" in nature and that the highest level of interaction can be achieved.

3.0 A LIBRARY OF INTERACTIVE FUNCTIONS

As part of an experiment in interactive design in architecture and architectural education (3) four basic interactive functions: EDIT, TABLE, WD and QUEST (with auxiliary functions NUM, WOR and CKQUEST) were written in APL to provide a general purpose library of interactive functions. Through a judicious use of these functions a programmer can provide a high degree of interaction between the user and the system which represents the problem solving algorithm with a minimum of effort.

3.1 The Function EDIT

For high level interaction, the interface should, with some latitude, be able to recognize words. Some words contain a large degree of redundancy which would involve extensive and expensive recognition functions. There are many possible ways of writing "bedroom 2" , e.g. "bed 2", "bedroom 2", "bedroom2", "bedroom-2", "bed-2". The real information, as far as differentiating bedroom 1 from bedroom 2, or from the bathroom, is contained in the type of room and its number. The method of word recognition used in the function QUEST, to be discussed later, uses only four characters from each word. The four characters are the first one and the next three that are not a, e or i. Experience has shown that this is usually sufficient for precise recognition. However, using this method, it seems that one could never get unique recognition from the words "bedroom1 bedroom2" since their compressions would result in "bdro bdro". The same problem can occur in other areas where these interactive programs can be used, e.g. in electronics; resistor 1, resistor 2, capacitor 1, transistor 8, etc. The need has arisen,

therefore, for a general-purpose function to eliminate redundant groups, if they occur. Such a function would free users from writing special functions for unique interface problems, when using interactive programs.

EDIT eliminates redundant groups of characters, not single characters, such as spaces or commas. It will eliminate multiple use of single characters, but with complete elimination rather than "all bar one" elimination.

The function EDIT will eliminate groups, such as "room", from a literal vector "bedroom1 bedroom2" to yield "bed1 bed2" which after further compression, for word recognition, would become "bd1 bd2". The final expression contains all the useful information - room type and room number. Some problems can occur because EDIT will remove any group which contains any of the characters to be removed. If we keep to the example of eliminating "room" and if the vector "broomroom" were to be edited, the result would be "b". This may be satisfactory for word recognition, depending upon the other words used. However, apart from problems of the type mentioned above, EDIT can be of great assistance to the interactive users in achieving a more fluid interface, without the need for extensive checking functions.

EDIT is a defined dyadic function with an explicit result and is of the form:

```
J EDIT FF
```

where:
J is the vector of any permutation of the redundant characters to be removed and FF is the vector or array from which the redundant characters are to be removed.

For maximum flexibility it will handle as its right argument a literal scalar, vector or array. Some examples are given below:

```
Example 1
Array to be stripped of "room"
   F
BEDROOM1
BEDROOM2
BEDROOM3
BATHROOM
KITCHEN
LAUNDRY
Using EDIT to remove "room" from an array of room-
names
        'ROM' EDIT F
BED1
BED2
BED3
BATH
KITCHEN
LAUNDRY

Example 2
Using EDIT in an electronics application
   E
RESISTOR3
RESISTOR8
RESISTOR45
RESISTOR67
'ISTOR' EDIT E
RE3
RE8
RE45
RE67
```

EDIT is a one line defined function in APL. The use of EDIT will be further demonstrated in Section 3.4 and 4.0.

3.2 The Function TABLE

During the development of the interactive part of the system it was found that strings (vectors) of literal characters needed to be handled and processed. Whenever input had to be processed there was always a problem of differentiating literal word or number groups from each other.

An associated problem with processing the user's response is to determine what a word is. What are the word delimiters? In most cases it is the presence of a space, but in some cases multi-word inputs may be required, where the vital information is contained in the use of both words. Such a case does exist if the user is given the freedom of

using names such as "bedroom 1" or "lounge room" in his input. If this is so, then some other delimiter is necessary.

In using QUEST, it is much easier to check for word recognition by indexing an array made up of the user's response, against an array of the function's vocabulary. By using the APL inner product, it is possible to check for exact word recognition and its location in the vocabulary in one operation. A similarly simple check, using the outer product and dyadic transpose, can be made for partial recognition. The great power of APL lies in its ability to handle arrays.

The use of the inner and outer products can, in many cases, eliminate the need to loop and so produce a considerable saving in processing time.

It is apparent that, with the need to process words as opposed to sentences and with APL's versatility with arrays, the best way to handle literal inputs was to assemble the input into an array. The size of the array would depend upon the number of words in a sentence and the length of the longest word. The result was the function TABLE which takes the form

J TABLE A

where
J is the word delimiter (a scalar) and A is the literal vector to be assembled into an array (table). Some examples are given below:

Example 3

```
 ' ' TABLE 'BEDROOM1 BEDROOM2 BATHROOM KITCHEN'
BEDROOM1
BEDROOM2
BATHROOM
KITCHEN
```

Example 4

```
 ' ' TABLE 'I WANT TO GO HOME'
I
WANT
TO
GO
HOME
```

Example 5

```
 E←' ' TABLE '1 2 3 4 5 6'
1
2
3
4
5
6

 F←',' TABLE 'ONE=, TWO=, THREE=, FOUR=, FIVE=,
                                              SIX=,'
ONE=
TWO=
THREE=
FOUR=
FIVE=
SIX=
```

Catenation of E and F:

```
     F,E
ONE=    1
TWO=    2
THREE=  3
FOUR=   4
FIVE=   5
SIX=    6
```

Example 6

```
 ',' TABLE 'BILL SMITH, PETER JONES, HARRY BROWN,
                                          B. GREEN'
BILL SMITH
PETER JONES
HARRY BROWN
B. GREEN
```

TABLE is a one line defined function in APL.

3.3 The Function WD

As mentioned in Section 2.1, there should be a varied response by the system to the user's input. The responses should be different and should not appear too contrived or repetitive. It is relatively simple to substitute a synonym for a word or phrase that is commonly used in the interactive process. A substitute word could be selected from either a vector or an array of suitable words. Here, TABLE is used to generate arrays from which the suitable words are indexed.

Some comments on word selection should be made first. The substitute word should show variety, and, above all, the word should mean what was intended by the original word. To some extent, the

context will change the meaning of the substitute word, so it is important that words selected for use as alternatives should be carefully checked before use for meaning.

In high level interaction, as much variety as possible should be used in constructing responses, but without the loss of meaning that can sometimes occur with general-purpose word replacement functions. The vocabulary should not be too intensely colloquial, otherwise the repetition will be more easily detected and the user will become weary of clever turns of phrase.

One drawback with the use of substitute words is that the form of the sentence always remains the same and in a process which involves the repeated display of the sentence, the contrived nature of word substitution may concern some users. In some cases, it may be possible simply to alter the sentence by randomly inserting or omitting different phrases.

The function WD was developed in an attempt to produce a general purpose function which would provide these features. It takes the form:

J WD X

Where
J is the numeric control scalar and X is the literal vector to which the substitute word is to be catenated. X can be a scalar or it can be an empty vector, if nothing is to follow the word.

J can be any number, but the function only responds to numbers between 0 and 120. WD responds in word selection to the "tens" part of the number entered. The "units" part of the number is converted to a binary radix and this is used to control other aspects of the word selected. This second control gives a choice of 4 unique controls because of the binary radix (i.e. 0 2 4 and 8). If it were intended to introduce more variety, then the actual digit could be used for control, in which case, 10 alternatives for each word would be available.

The following table gives values for J which select similar words to the words shown:

J	WORD	
10	alter *	ø
20	right	
30	good!	
40	enter *	ø
50	error	
60	check	ø
70	try	
80	want *	
90	entry	
100	help	
110	understand	

The words marked * are verbs and are supplied as infinitives with the J used, as the present participles if 1 is added to J and as the past participles if 2 is added to J.

Further variety to the words marked ø can be achieved by adding 8 to J. This will cause expressions such as "please", "now then", etc. to precede the word selected.

The word selected by J=20 can be followed by a comma, if desired, by specifying J as 21.

Some examples are given below:

Example 7

```
    'DO YOU', 80 WD 'TO', 10 WD 'ANYTHING?'
DO YOU NEED TO CHANGE ANYTHING?
```

Example 8

```
    20 WD 'WARREN'
RIGHT WARREN
```

Example 9

```
    40 WD 'YOUR NAME'
ENTER YOUR NAME
```

Example 10

```
    41 WD 'YOUR NAME'
KEYING IN YOUR NAME
```

Example 11

```
    42 WD 'YOUR NAME'
ENTERED YOUR NAME
```

Example 12

```
    48 WD 'YOUR NAME'
NOW KEY IN YOUR NAME
```

WD is a 17 line defined function in APL.

3.4 The Function QUEST

If one wants high level interaction with users who "talk" to the system, then some cognizance must be taken of how the user "talks." Section 1.0 described some of the underlying problems of input. In order to allow the greatest flexibility numeric input (the quad input in APL; F-or I-formatted input in Fortran) should be discarded in favour of literal input (the quote-quad in APL; A-formatted input in Fortran).

Obviously, the problem is not solved by simply replacing quad with quote-quad. The problem really begins at this point for the programmer. All numeric entries, in quote-quad, are simply character strings and these must be differentiate d from each other, from words and from whatever other characters are entered by the user. It is relatively simple to devise a function to convert a character vector, representing a number, into its numeric counterpart, but word recognition is more difficult.

The eight points listed in Section 2.1 have been combined in the function QUEST which is essentially a question-asking and answer-recognition function.

Its flexibility and scope need considerable description; it is one of those functions which should be experimented with for some time prior to its use in other functions. It is a problem oriented function which is moulded to suit the problem involved by use of its control facilities. It processes literal inputs, representing numbers and/or words and returns a numeric vector after interpretation and checking.

QUEST is a recursive function and takes the form:

J QUEST FF

Where J is the control argument and FF is the literal vector forming the text of the question or command to be posed to the user. J can be a scalar or a vector. If it is a scalar, then, certain primary controls are set up, controlling the types of responses accepted by the system and also controlling the system's responses to the user's input.

If J is a vector then its length has significance, as well as its value. J has significance for lengths up to four. Any further elements in J are ignored, so far as control is concerned. Therefore, QUEST can take the form:

(JA JB JC JD) QUEST FF

where JA is the primary control element with the following significance:

JA	EFFECT
512	Allows the "further question" facility
256	Allows the "special word" facility
128	Bars numbers not in index list of vocabulary vector
(64	Is used for recursive control)
32	Allows non-integers
16	Allows vectors
8	Allows negative numbers
4	Allows further controls JB, JC and JD
2	Allows "help"
1	Allows "yes" and "no"
*	Allows display of non-recognition warning. (* = any decimal fraction less than 1 but greater than 0)

Once the programmer has decided upon the type of entry he wants from the user, he can then set JA by adding together the values for JA which will give him the facility he needs. Default conditions exist in the event of non-specification. If JA is not set with 16 in it then only scalars will be accepted, etc. If, for example, the programmer wants the user to be able to enter vectors, but all the elements of which must be integers, although, negative integers would be acceptable, then he would set JA = 16 + 8 = 24. If he wants the user to be able to ask for "help", in the event of his having difficulties, then JA would become 26. Similarly, if the programmer wants the user to be able to exit from the function (or part of the function) by using words such as "stop" or "finish", then JA would become 26+256 = 282.

If 4 is added to JA, then QUEST will consider the significance of the next one, two or three elements of J, i.e. JB, JC, and/or JD, where:

- JB Sets the floor (minimum) of a scalar or vector,
- JC Sets the ceiling (maximum) of a scalar or vector,
- JD Sets the length of a vector.

The only proviso here is that, if JC, then JB must exist, i.e., if only the maximum value of the entry is important then the floor controller, JB, must be set to insignificance. Similarly, if only the number of elements entered is vital, while the range of values is inconsequential, then both JB and JC must be set to insignificance.

Thus, continuing the example above, the following specification of QUEST:

286 30 216 3 QUEST 'What numbers?'

would have the following effects: The user must enter a 3-element vector, with values within the range of 30 to 216, with all the elements being integers. The user can ask for "help" or can demand to "stop" (if the special word vocabulary was set up using words similar to "stop").

Before discussing the system's responses and the responses of QUEST, we should discuss the prerequisites that are necessary for QUEST's operation. QUEST uses auxiliary functions NUM and WOR, as well as TABLE, EDIT and WD. WD is used to add variety to the terminal's responses during QUEST's processing, if necessary. Users may wish to dispense with this facility, in which case, the responses can be rewritten with fixed forms. The other functions are necessary for QUEST to operate.

The function NUM converts all entered "numbers" (character strings) into numbers. The function WOR interprets entered words and for this to evaluate it needs a vocabulary against which to compare the entered words. WOR uses a stripped vocabulary, QN, which is an array with as many rows as words in the full vocabulary QN but only four columns. The stripped vocabulary is either prepared by using a function or by setting up the array. QUEST indexes QN (a vector of numbers which corresponds to each word in the full vocabulary) for recognized words and its output is numeric and corresponds to those elements of QN which represent the recognized words in QN (i.e. ultimately in the full vocabulary QN).

As described above, the functions EDIT and TABLE both require left control arguments. Both functions are used in QUEST and in the form:

J1 EDIT J2 TABLE A

(processing occurs from right to left) where A is the users response and J1 and J2 must be specified by the programmer. Flexibility is given to the programmer by allowing him to specify the word delimiter he wishes to use and by giving him the facility to eliminate redundacies from the user's input prior to processing by WOR. Therefore, J1 is the redundancy eliminator and J2 is the word delimiter. If the programmer wants a space to be the word delimiter then J2 is set to ' '. If multi-word descriptions are to be used and a comma is needed to delimit words then J2 would be specified as ','. Since J1 must be specified and since in some applications redundacies will not occur, J1 can be set to an unlikely character, such as '!'

If the "special word" facility is used, then the programmer must specify a list of special words which the user may use. The list of "special words" is specified as SW has as many rows as words in the list and four columns. SW can be set up using TABLE. If the "special words" were "stop", "end", "finish", and "go", then SW would take the form:

```
stop
end-
fini
go--
```

where - indicates blanks in the array. If any of these words are recognized (all the input "words" are compressed to four characters) then QUEST terminates and returns a certain result.

Similarly if the "further question" facility is used then a literal vector FQ must be specified. FQ contains the text of the question which will follow the question FF. If the user anticipates FQ and does not answer "yes" then FQ is not posed and the users input is taken as his having anticipated FQ and answered it. If he answered "no" to FF (and FQ is available) then an exit is taken from QUEST.

In both the case of SW and FQ, these need not be specified unless they are needed (i.e. JA=256 or 512), since QUEST branches around unused statements which are, therefore, not compiled.

Since QUEST is used internally in other functions

FQ and SW can be localized to the function in which they are being used. Further, if QUEST is used more than once in a function then SW and FQ can be respecified as often as needed. The same applies to the vocabulary (QN), stripped vocabulary (QN) and vocabulary vector (QN).

The operation of QUEST is arranged so that there is a heirarchy of control. If all the SW, FQ, "help", and "yes/no" facilities are available then SW is checked for firstly. This means that a response, such as, "yes, I want to add a room", is interpreted as the need to add a room, rather than a need to alter the input. Similarly a statement such as, "yes, but I want some help first," will cause a branch to help routine.

From the foregoing, it should be apparent that QUEST will return a numeric vector which is its interpretation of the user's character input. The input can be mixed alphameric with words and/or numbers and converts the "numbers" into numbers and converts recognized and partially recognized "words" into numbers from the vocabulary vector. QUEST checks the interpreted input against the control criteria and then, after user confirmation of QUEST's interpretation, (if necessary), a numeric vector is returned as the explicit result.

For special conditions a specific vector is returned. The vector is a four element vector of 1's and 0's which can be used for branching.

QUEST is a recursive function which will successively re-evaluate until the user has complied with the control condition either by supplying the correct input or by using one or other of the available options.

Below are some examples of the use of QUEST:

The following is the text of the right argument of QUEST:

```
     FF ← 'TYPE IN A NUMBER'
```

The * indicates the user's response.

Example 13

```
    8 QUEST FF
TYPE IN A NUMBER
*5
 5
```

Example 14

```
    8 QUEST FF
TYPE IN A NUMBER
*4 5 6
      ONLY THE FIRST ENTRY WILL BE USED
 4
```

Example 15

```
    8 QUEST FF
TYPE IN A NUMBER
*HELP
THERE IS NO AID HERE, JUST ANSWER THE QUESTION.
TYPE IN A NUMBER
*I WANT THE NUMBERS TO BE ¯45.7 and 56.8
     ONLY THE FIRST ENTRY WILL BE USED.
     THE FRACTIONAL PART WILL BE IGNORED.
¯46
```

Example 16

```
    8.98 QUEST FF
TYPE IN A NUMBER
*I TOLD YOU THAT I WANT THE NUMBER TO BE ¯45.7 ÷ 78
I CAN'T COMPREHEND PART OF YOUR ANSWER
     ONLY THE FIRST ENTRY WILL BE USED.
     THE FRACTIONAL PART WILL BE IGNORED.
¯46
```

QUEST is a 38 line defined function in APL.

5.0 DISCUSSION

These four interactive functions plus their auxiliary functions can be easily used by a programmer to provide an interface between the user and any solution algorithm in a design process. The functions are problem independent and form a group of basic functions available for general programming applications. Their successful development has been largely dependent on the capabilities of APL. Listings of these functions are available from the authors.

The problem of spatial layout design has been tackled interactively using these functions as the interface with considerable success (3).

6.0 ACKNOWLEDGEMENTS

This work is the result of a joint project between the Department of Architectural Science, Sydney University and the Systems Development Institute of IBM (Australia), Project No. 71-039; additional support has come from the AVCC-SCREEM and the URG.

7.0 REFERENCES

1. McKINLEY, J.T.: Conversational Techniques Developed for Remote Access Computer-Aided Design. Computer-Aided Design, I. Elect. Eng., London, 1969, pp. 162-170.

2. ...: APL/360 User's Manual. IBM Corp., CH20-0683-1 1969.

3. GERO, J.S., JULIAN, W.G. and HOLMES, W.N.: The Development of a System for Heuristic Optimization of Topological Layouts Using High Level Interaction. Department of Architectural Science, University of Sydney, Computer Report CR22, 1973.