A DATA SUB-LANGUAGE FOR CASUAL USERS

Ian A. Macleod
Queen's University at Kingston

Abstract

To enable users to effectively exploit a complex computer system, it is necessary to construct a powerful yet usable interface. Conventional formal dialogues have the disadvantage that they are virtually unusable by casual or untrained users. Free format systems, on the other hand, tend to be either very simplistic or else require extremely complex parsing mechanisms. Even in the latter case they can do little better than understand a restricted subset of a natural language thereby offering little advantage over the usual formal language systems.

In this paper we show how a formal language processor can be relatively easily used to develop a system which permits comprehensive user interfaces to be constructed yet at the same time, through interactive techniques, can effectively permit access to casual users. The system is implemented on a PDP 11/45 under the UNIX operating system.

SOUS-LANGAGE INFORMATIQUE POUR UTILISATEURS OCCASIONNELS

Résumé

Afin de permettre aux utilisateurs de tirer le meilleur parti d'un système informatique complexe, il est nécessaire de construire des interfaces offrant de multiples possibilités, mais qui soient en même temps utilisables. Les dialogues formels classiques présentent l'inconvénient d'être à peu près inutilisables par les usagers occasionnels ou inexpérimentés. D'autre part, les systèmes à structure non imposée ne sont guère plus efficaces; ou bien ils sont trop simplistes, ou encore ils nécessitent des mécanismes extrêmement complexes d'analyse grammaticale. Même dans ce dernier cas, les systèmes ne peuvent guère s'élever au-delà de la compréhension d'un sous-ensemble restreint de la langue naturelle, de sorte qu'ils offrent peu d'avantages par rapport aux langages formels habituels.

La présente communication montre comment il est relativement facile d'utiliser un compilateur de langage formel pour développer un système permettant d'une part d'établir des communications extensives avec l'utilisateur et d'autre part, grâce à des techniques interactives, de donner accès aux usagers occasionnels. Le système est mis en oeuvre sur un PDP 11/45 avec le système d'exploitation UNIX.

A Data Sub-Language for Casual Users

Ian A. Macleod

Queen's University at Kingston.

INTRODUCTION

In developing computer systems which are intended for use by people who may have no knowledge of computers and programming languages and who cannot be expected to have any great desire to acquire such knowledge, there is a severe problem in designing an interface which will enable them to make effective use of a complex computing system. The obvious example is that of information retrieval systems. In this case there typically exists a highly trained and intelligent user population hopeful of retrieving information but unwilling to cope with such subtleties as are normally imposed upon them by a programmed system. In this type of situation it is necessary to provide an interface which is sufficiently rich to enable the user to exploit fully the resources of the system, but which is not so computer language oriented as to provide a serious technical barrier to the potential user, untrained and inexperienced as he is in the mystical ways of computers. Management information systems, library retrieval systems and even job control languages are examples of systems intended for the potentially casual use by a broad spectrum of people. Here it is highly desirable to fit the computer system to the user. If this is not the case we frequently find that systems either are used extremely ineffectively, as is often the case with programmers and operating systems, or at worst are not used at all by people who could benefit greatly from them.

The obvious type of interface is based on natural language. However, even among humans, English commands and queries can be vague and ambiguous. Witness for example many of the computing industry's manuals. In any case it has not yet been found possible to implement a recognizer which is capable of effectively understanding English input. An alternative to full English is some subset of the language. Processors, such as REL,[1], which can analyze such restricted English dialogue have existed for some time. They have the major disadvantage that the user must either learn to cope with the limitations of such systems or run the serious risk of being misled into overestimating both the degree of understanding possessed by the computer and the extent of the information implicit in the data base itself.

Programming language types of interfaces are fairly common. Typically these interfaces have a rigid syntax where the format of the queries is oriented towards a particular database. These systems, if they are constructed with sufficient care, can be extremely useful. Queries can be formulated with precision and the language can be designed so as to reflect the information implicit in the database. However, while they are relatively easy to learn, particularly for a person with some previous computing experience, they do require a certain amount of training - certainly more than the casual user would normally be prepared to endure. Also, such systems frequently have the characteristic of displaying unhelpful messages such as "ILLEGAL COMMAND" or, worse still, "SYNTAX ERROR". Few things are more likely to deter a potential user than a blunt refusal by the system to have anything to do with his attempts to communicate.

To simplify the interface, a menu approach can often be used. In this case, at each stage in the process of building a query or command the system provides the user with a list of alternatives from which he selects one or more simply by typing in a number, or, in the case of some displays, by selecting the appropriate command by means of a light pen. Another technique is to have the system query the user. This type of interface may be used either as an alternative to or in conjunction with the menu approach. Such dialogues can be designed so as to be virtually foolproof but for the experienced user they can be intolerably slow, particularly, but not exclusively, on a typewriter terminal. Furthermore these types of dialogue can be very inflexible.

The technique proposed here is to combine two approaches. A basic formal language interface is provided. When the user responds to the system it attempts to understand the response. If it can, that is if the command or query is "syntactically correct", it then processes the communication appropriately. If it cannot, it gleans what information it can from the input and then attempts to complete the request by querying the user. Thus the person with experience makes use of the system with no unnecessary interaction while the inexperienced user interacts with it through a combination of menus and computer-initiated queries. Hopefully, as the novice gains experience, the degree of interaction lessens. This type of approach appears to be ideally suited to query languages which normally operate in an interactive environment and also, typically, have a relatively simple grammar.

THE MISTRAL LANGUAGE PROCESSOR

Mistral is a language processing system designed to provide a vehicle for the implementation of user interfaces. It is

both extendible and interactive. Dynamic syntactic and semantic extensions are permitted. The full capability is described elsewhere, [?], but briefly it provides means of defining new syntactic constructions, or language units, as for example by the following command:

```
Define language-unit-name;
nonterminal = syntax-specification;
semantics;
```

Here the syntax specification is provided by a meta-language based on an extended version of BNF and the semantics are written in terms of existing language constructs. Once a new language unit has been defined, it immediately becomes available for execution or for incorporation into the semantics of further language units. In the context of this work, one of the more interesting aspects of Mistral is that it is based upon an implementation of Earley's parsing algorithm, [3]. One of the features of this algorithm is that it is built around a top-down predictive parse where all possible parses are carried along simultaneously. A parse is successfully completed when the entire input string has been processed and a single syntax tree has been produced. In the eventuality of a syntax error occurring, a number of partially built trees exist. These represent all the possible parses up to the point at which the error occurred. Our approach here assumes that at least one of these trees represents the beginning of a potentially correct syntactic interpretation of the user query. It is then probable that the parse can be continued successfully by requesting information from the user. The type of information required is determined from the partial trees.

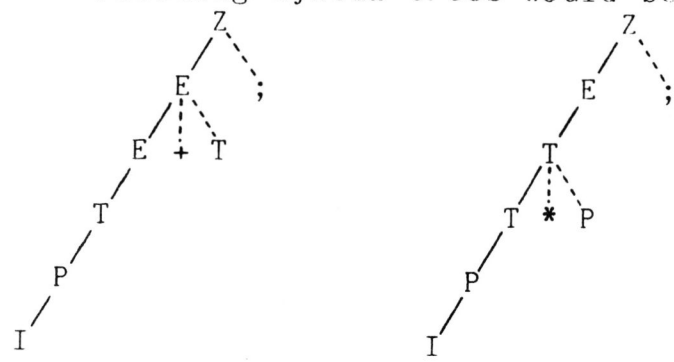
For example take the following simple grammar:

```
Z -> E;
E -> T;
E -> E+T;
T -> P;
T -> T*P;
P -> (E)
P -> I
```

Then for the following input string:

```
I (I + I);
```

the following syntax trees would be produced:



The dotted lines denote incomplete branches.

Obviously at this point the parse can proceed no further until some missing information can be supplied. In this trivial example it is apparent that either a "+" or a "*" has been omitted. In an interactive environment we can determine which by querying the user.

The actual error recovery algorithm is based upon one suggested by Irons,[4]. When a syntax error occurs there will exist a partially processed string of the form:

uTt

where "u" is the portion so far parsed, "T" is the symbol at which the parse failed and "t" is the remaining unprocessed string. The following steps are then performed:

(i) A list is constructed of all the symbols in incomplete branches of the tree, (T, P and ";" in the example above).

(ii) The first symbol of the string "Tt" is repeatedly examined, discarding the first symbol if necessary, until a string is found such that for some symbol U in the list:

U =>+ T... (if U is a non-terminal),

or

U = T (if U is a terminal)

(iii) By examining the incomplete branches before the branch containing U, it is determined what information is missing. Where the missing information represents one of several choices, the appropriate string to be inserted is obtained by querying the user.

(iv) The parse then continues with any unwanted parse trees discarded.

Note that this approach assumes that the partially parsed string "u" has been correctly interpreted. In practice this may not always be correct. However our main aim is not to develop a deterministic technique, but rather an approach which has a high probability of success. Initially the system is being applied in the development of interfaces for document retrieval systems where the possibility of a disastrous mis-interpretation of a user query is in any case not high.

EXAMPLES

Take for example a small information retrieval language whose syntax is specified as follows.

```
IRCOMMAND = "FIND" NUMBER "ON" SEARCHEXPR
IRCOMMAND = "LIST" NUMBER;
SEARCHEXPR = STRING ("AND"/"OR" STRING)...
NUMBER = "ALL"/INTEGER
```

Examples of these commands are the following:

```
        FIND ALL ON COMPUTER AND RETRIEVAL;
        LIST 10;
```

Suppose a naive user types the request:

```
     GET ME EVERYTHING YOU HAVE ON INDUSTRIAL POLLUTION;
```

The following syntax trees would be generated:

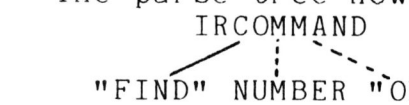
The only incomplete symbol which matches is "ON" so a FIND command is assumed and the unrecognised symbols before the "ON" are discarded.

```
      SYSTEM: Do you wish to use the FIND retrieval
              command?  Type Y or N.
      USER:   Y
```

A "FIND" symbol is inserted and the parse continues following the display of the amended query as:

```
      FIND ON INDUSTRIAL POLLUTION
```

The parse tree now becomes:



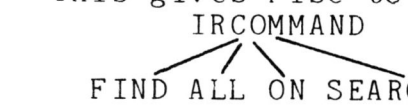
The incomplete branch is caused by the non-terminal NUMBER.

```
     SYSTEM: Do you want: 1 - all relevent documents or
             2 - some specific number?  Type 1 or 2.
     USER:   2
```

The modified command is now displayed as:

```
     FIND ALL ON INDUSTRIAL POLLUTION
```

This gives rise to the syntax tree:



Unfortunately Mistral requires all its literal strings to be in quotes. This would give rise to the following possibly annoying dialogue.

```
     SYSTEM: Type in a single word which describes the
             type of document in which you are interested.
     USER:   industrial
     SYSTEM: FIND ALL ON "INDUSTRIAL";
             Are there any other words which describe the
             documents you want? Type Y or N.
     USER:   Y
     SYSTEM: Type in the next word.
     USER:   Pollution
     SYSTEM: Should documents contain references to both
             these terms or just one? Type Y or N.
     USER:   Y
     SYSTEM: FIND ALL ON "INDUSTRIAL" AND "POLLUTION"
             Are there any other words which describe the
             documents you want? Type Y or N.
     USER:   N
```

The parse now concludes sucessfully and is processed. In this case the system leads the user through every possibility and each time it modifies the query it displays the partially completed query. Thus the user is able to learn, at his own speed, the format in which the system expects its commands. While this level of query processing is extremely tedious for an experienced user this is not so in the case of an inexperienced user. The amount of dialogue which will ensue will depend on the amount of information the user has correctly supplied.

For example suppose a slightly experienced user types:

```
GET ALL "INDUSTRIAL" AND "POLLUTION";
```

The system will through a similar but much shorter dialogue modify this to read:

```
FIND ALL ON "INDUSTRIAL" AND "POLLUTION";
```

IMPLEMENTATION

The system is implemented by attaching a recovery procedure to each language unit where this is desirable. This is done by a command of the form:

```
Error recovery-routine language-unit-name;
```

When a syntax error occurs in the named language unit the recovery routine is invoked. In our current implementation recovery routines are written in the implementation language which is C, [5], a well known systems programming language.

A number of special functions have been implemented to allow the parse trees to be traversed and modified. These include:

(i) reportback; Invokes the error routine of the parent language unit in the tree, (or its parent if no such routine exists).

(ii) next; Invokes the error routine of the language unit in the next possible parse. If there is no next alternative the effect is the same as a reportback.

(iii) parent; Returns the name of the parent language unit.

(iv) insert; Inserts symbols into the string being parsed.

(v) atom; Gives the position of the atom within the syntax at which the parse failed.

(vi) parse; Continues the parse with the presumably amended input string.

(vii) abort; Discontinues the parse.

For example we might develop a dialogue for a part of the language given above as shown below. First the language is augmented with a production of the form:

```
Define root;
```

```
      command = ircommand
```

This allows a recovery routine to be associated with the situation in which no retrieval command is recognised. The names of the routines are then specified as follows:

```
      Error root rooterr;
      Error find finderr;
      Error number numberr;
```

Then the recovery routines might be represented by the following C routines:

```
rooterr()
 {
loop:
  printf("Do you want to 1. Retrieve, 2. List");
  printf("or 3. Neither? Type 1, 2 or 3");
  input = getchar();
  switch(input);
  {
   case '1':
            insert("find",1);
            parse();
   case '2':
            insert("list",2);
            parse();
   case '3':
            printf("try again");
            abort();
   default:
            goto loop;
  }
 }

finderr()
 {
  if(atom() == 1)
    {
loop:
    printf("Do you want to retrieve? Type Y or N");
    input = getchar();
    if(input == 'Y')
      {
       insert("find",1);
       parse();
      }
    if(input == "N")rooterr();
    goto loop;
    }
  if(atom() == 3)
    {
    insert("on",3);
    parse();
    }
 } /* no other possibilities */
```

```
numberr()
 { loop:
 printf("Do you want 1. All the documents
         or 2. a specified number? Type 1 or 2.");
 input = getchar();
 if(input == '1')
   {
    insert ("all",2);
    continue;
   }
 if(input == '2')
   {
    /* dialogue to read in and convert the string
       of digits typed by the user */
   }
 else goto loop;
 }
```

SUMMARY

It is by no means obvious that the approach outlined here is an ideal one. Nor is it obvious that an ideal approach does in fact exist. There are still the problems caused by the misinterpretation of a query which is syntactically correct but which is not at all what the user intended. This is not likely to be a severe problem in our own limited application to a document retrieval system but would probably be serious in a more general application. A further problem is that the design of satisfactory dialogues for more complex language than the one illustrated above will certainly be non-trivial. However we feel that this type of approach does at least provide some potential for improvement of most formal language interfaces. Its true worth can only be gauged in a real life application where some measurement of actual successfull error correction can be made.

REFERENCES

1. Thomson P,C. et al: REL, A Rapidly Extensible Language System, ACM 24th National Conference, 1969.
2. Avis, J. and Macleod, I.A.: An Extendible Interactive Language Development System, Technical Report, Queen's University, 1977.
3. Earley, J.: An Efficient Context Free Parsing Algorithm, Comm. ACM, Vol 13, 1974.
4. Irons, E.T.: An Error Correcting Parse Algorithm, Comm. ACM, Vol 6, 1963.
5. Ritchie, D.M.: C Reference Manual, Bell Laboratories, January 1974.