Implementation of a syntax-driven interpreter for data retrieval‡

By A. J. Fox and P. W. Edwards*

An account is given of the design and implementation of a syntax-driven interpreter. The application of this interpreter to various facets of a data retrieval project is discussed. Particular emphasis is placed on the role of syntax descriptions both for specifying the retrieval language and also for classifying the data. The main features of the retrieval language, CLIC, are presented.

(Received November 1968)

1. Introduction

Whilst syntax techniques have been widely employed for specifying and translating general purpose programming languages, less attention has been paid in this country to the use of these methods for other purposes, such as on-line interrogation or even the systematic description and processing of data. (For a comprehensive review, the reader is referred to Feldman and Gries, 1968.) In the present paper, we describe a program currently used at RRE to retrieve data on the attributes of integrated circuits (their names, makers, function, place of origin, type of packaging, cost, etc.) This program stemmed from a very early foray into conversational programming techniques by J. M. Foster (1967), which was probably the first query language program to embody the technique of 'lambda interpretation' described in this paper.

In summary, the data on integrated circuits is prepared by circuit engineers on a pre-printed form, punching onto tape is performed directly from the form, and thence, via a syntax directed input routine, the data is accepted by the computer and filed on magnetic tape to form the data base for the program. The computer, RREAC, is used primarily for the batch-processing computing service to the Establishment as a whole, but core-swapping arrangements have been made to enable users of the retrieval system to obtain immediate access to RREAC using an on-line teleprinter. We do not propose to discuss this side of things any further.

The interrogation program consists of a syntax analyser which accepts queries satisfying grammar rules appropriate to our particular fields of application. The grammar rules are data to the analyser, and may therefore be changed readily (see Section 2). The output from the analyser consists of a lambda-expression, which embodies substitution operators and calls of 'primitive procedures' which have already been incorporated in the ALGOL program (see Sections 3 and 4). The lambda expression corresponding to any input query is 'evaluated' by the lambda interpreter and results in the performance of the task demanded.

2. Actions embedded in syntax

For most computer applications of syntax techniques it is inadequate merely to recognise a sentence as being grammatically correct: it is necessary to associate with the analysed sentence a set of actions which one wishes to be obeyed. In our implementation the names of the requisite actions are embedded amongst the productions in the syntax specification.

Consider the example:

$$\langle A\rangle ::= a\alpha\langle B\rangle \mid b\beta$$

$$\langle B\rangle ::= c\gamma\langle A\rangle \mid d\delta$$

where the Greek letters denote the names of actions. The desired aim is that when a sentence *a c b* is analysed as an example of the class $\langle A\rangle$ then the actions $\alpha\ \gamma\ \beta$ will be obeyed. Similarly *b* should give rise to the action β and *a d* give rise to $\alpha\ \delta$. For the moment it will be assumed that actions will be stacked up for obeying as program at a later stage. The algorithm for a predictive analyser written to achieve these effects is outlined in Appendix 1.

It should be noted that this algorithm pursues all successful matches in an expansion with alternatives. It also avoids the necessity for back-tracking on the input stream when a terminal symbol mismatch occurs. Another feature of this system is the ability to embed the actions at any point in the syntax productions. This is more flexible than allowing actions to be placed only at the end of a production or with a phrase marker in an analysis tree. Even so further manipulation of the actions is necessary and will be described in the section on the interpreter.

The analysis algorithm as already described could prove slow in analysis and so the basic scheme was enhanced. Following standard compiler writing practice, identifiers and numbers are read in and assembled into complete words by a separate procedure. These input words are then passed through a vocabulary lookup process to link them with terminal symbols. The words in the vocabulary are all allowed to have an associated meaning which is treated in an analogous way to actions. The alterations to the basic analyser to incorporate these features are indicated in Appendix 2. The syntax itself was also optimised using the program SID (Foster, 1968) into a form which reduced multitracking.

3. Lambda interpreter

The actions produced by the syntax analyser are taken as a program stream to be interpreted. As the problem was initially envisaged, the output to the analyser could be arranged, by suitable positioning of the actions in the syntax, to be a reverse polish string. In practice the output string requires macro manipulation of both the data and the subroutine calls to achieve the correct

‡ Crown copyright. Reproduced by permission of the Controller of H. M. Stationery Office.

* *Ministry of Technology, Royal Radar Establishment, Malvern*

Downloaded from https://academic.oup.com/comjnl/article/12/3/225/363357 by guest on 19 April 2024

sequencing. This simple task is accomplished by making the action commands in λ-calculus (Landin, 1964). The stack manipulation facilities offered by this language are illustrated in **Fig. 1.** Here a complete syntax and interpreter specification is given for converting from cash expressed in £ s d into pennies. In this trivial example the primitive subroutines could have been embedded directly in the syntax to produce a reverse polish output stream. This rendering of the example also provides the opportunity to display the exact input conventions used on RREAC. A natural bracketting notation was adopted for the actions with a comma separating the bound variables from the body of a λ-expression. The metasymbols chosen are: for λ and * for the apply symbol.

```
Vocabulary
stop = . ;
sla  = / ;

Primitive subroutines
mult
add

Actions
alsd = (( : l s d, 240 l mult * 12 s mult * add d add *) *)
asd  = (( : s d, 12 s mult * d add*) *)

Syntax
sentence=cash stop;
cash     = integer sla integer sla integer alsd,
           integer sla integer asd,
           integer;
```

Fig. 1. Specification of syntax map. In the syntax and vocabulary specification comma separates alternatives, semicolon terminates productions.

An initial set of primitive subroutines was provided whose actions are concerned with the interpretive system itself. The syntax analyser is the most important member of this set. The requisite primitives for the data retrieval tasks were then added. The primitives together with the interpreter were all programmed in ALGOL with list processing extensions.

The λ-calculus has certainly provided a convenient way of stack manipulation. It also has the advantage that the interpreter is a short and simple program to implement. However the action language has taken on a more important role than was originally conceived for it. The use of functional parameters, recursion and also the setting of identifiers in the environment of the λ-calculus have all been found useful, though this greater use of an intermediate language has emphasised the inherent difficulties in programming in an obscure functional language. The situation has been slightly alleviated by providing useful primitives for use in the λ-calculus itself. Thus the intermediate language bears a strong resemblance to LISP S-expressions. In future work in this field we shall use an expanded intermediate language. The major features which this should have are declarations and sequential statements. The extent to which fragments of the intermediate language itself can be precompiled will be investigated.

4. System organisation

The three basic ingredients of the programming system are the interpreter, the analyser and the primitives. The two major types of data are the various vocabulary and syntax specifications and search data. The total system organisation is indicated in **Fig. 2.**

The control of the system is deemed to lie with the interpreter except when it calls up a primitive. Each primitive must return control to the interpreter. All primitives take their input parameters from the evaluation stack of the interpreter. Most primitives plant output back on the evaluation stack. The analyser is unique in that it alone can plant further actions on the control stack of the interpreter.

Return of control from the analyser to the interpreter is achieved by one special action. This allows the system writer the choice as to whether the actions should be evaluated as they are encountered or alternatively whether they should be stacked up until the end of the sentence. Return during the analysis of a sentence can be used to introduce a measure of context dependence. This is achieved during the recall period by assigning a value to a variable within the environment stack of the interpreter and then recalling the analyser within the environmental scope of that variable. The effect of the latter technique is to plant information for use in subsequent actions generated within the scope of that call of the analyser. In practice return is performed only at a few places during the analysis of a sentence. When such a return is made the state of all the prediction piles in the analyser is maintained between activations. The interpreter itself, however, does not allow parallel processing; thus the return from analyser to interpreter can only be performed when there is a unique action stream.

Ideally all the primitives should be pure procedures. However the lack of declaration and explicit assignments within λ-calculus has meant that communication via

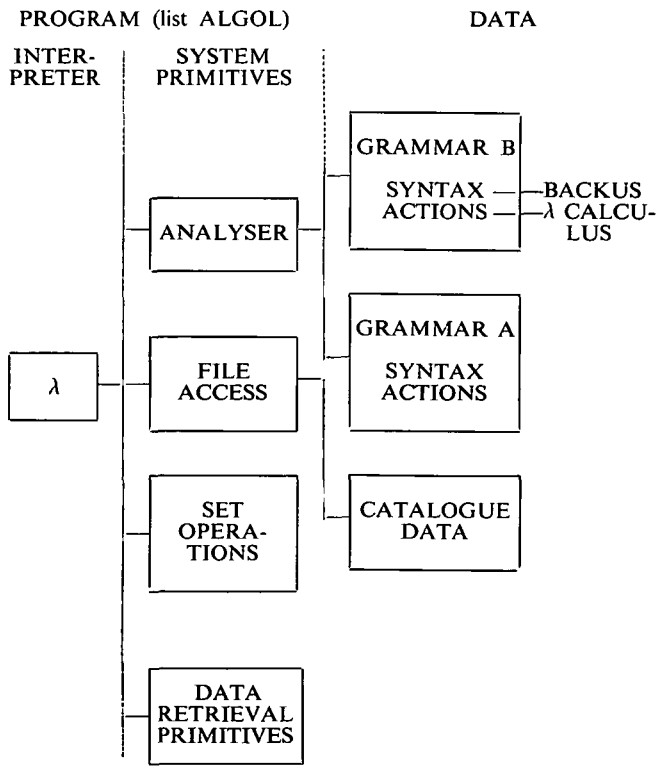

Fig. 2. Interpreter scheme

Downloaded from https://academic.oup.com/comjnl/article/12/3/225/363357 by guest on 19 April 2024

globals within ALGOL has often been preferred. Again, a more convenient intermediate language will overcome this difficulty.

5. The data retrieval task

Let us now consider the data retrieval task to which the interpreter was applied. The design aim was to provide an on-line retrieval system, a decision which deeply affected the organisation of the data files. A commonly used format for storing data is a set of separate records with each record containing a list of properties about a single item (in this context a property is considered to be an attribute name together with a value). With this format it is necessary to search all the records in the data base in order to answer any query. This is an inefficient process whenever the number of properties in the enquiry is significantly less than the number of properties in the record. In such a case, economical use of the computer can only be achieved by batching enquiries and answering them all in a single sweep through the file. Naturally this implies an off-line retrieval system. In an on-line retrieval system, where a rapid servicing of enquiries is necessary, a series of inverse files must be constructed, with each file applying to one particular attribute. Such a file consists of the values pertaining to that particular attribute together with the references to the records in the main file where they are contained; the complete file is sorted in order of the value. It still remains necessary, however, to retain the serial record file so that information may be presented about the records selected from the inverse files.

The program for creating and maintaining both types of files was written using the interpreter scheme. The input language for this program is syntax specified but is trivial. In essence it consists of one of the commands *enter*, *delete* or *amend*, together with sufficient information to identify the data item being altered. In the case of *enter* or *amend*, there then follows the value of the new data item which is itself syntax specified. If necessary any such data value can be annotated by means of a free text footnote. The form in which the data is organised within a record leans heavily on syntax techniques as will be described in the following section.

The retrieval program also makes extensive use of syntax techniques both for translating the query language and for directing the print-out of the more complex values. The query language is described in Section 7, followed in Section 8 by brief details of the actions used in the interpretation. It would have been quite possible to write a single program with a comprehensive language to embrace both the updating and retrieval tasks. However the nature of the data capture fits naturally into an off-line system with a batched set of updating commands, so the two tasks were programmed separately on RREAC. The data input and the data retrieval program are of comparable size, each one being approximately 20,000 compiled orders in length. In addition each program has a set of syntax maps and action details associated with it.

6. The description of data

The concept of a record as a list of properties has already been introduced. Often, attribute names are common to a large sequence of records and, in such cases, they may be held separately from the values. The records are thus reduced to a list of values in a contiguous area of store. A system must then be developed for describing attribute names and linking them with the values. The syntax techniques already described provide a way of accomplishing this task. We shall now enlarge on this observation and, in order to bring out the main features, we shall compare it with the well known COBOL system.

Let us first consider how attribute names may be introduced. In COBOL basic attributes are listed together with a description of their associated type of value. These basic attributes may then be collected under generic attribute names to give a tree-like grouping. Such a mapping of attribute names on to values can be trivially represented by a context free phrase structure grammar (CFPSG). In this grammatical (or syntactic) description, class names fulfill the role of generic attribute names and terminal symbols correspond to value descriptions. The data description in COBOL also allows one to repeat an attribute a fixed number of times using the OCCURS clause. There is no elegant way of achieving this effect in a syntax description, one must simply list the attribute name the requisite number of times.

However, the very simple recursive definition

$$\langle \textit{repeated name} \rangle ::= \langle \textit{name} \rangle \langle \textit{repeated name} \rangle \mid \emptyset$$

will allow an indefinite number of repetitions. As a consequence of the variable repetition feature, the record size will expand or contract according to the volume of data.

The recursive definition of a list of names is just one example of how the use of an empty alternative may be used to form a compact record. This must be contrasted with the COBOL practice of filling out the record to a fixed length, using dummy values if necessary.

A COBOL record may be compacted to a limited extent by means of the DEPENDS ON clause, though this device may only be used at the end of a record. This limitation must be contrasted with the flexibility of the recursive syntax definition which may be used at any point in the description of a record and can be nested to an arbitrary depth. The syntax definition given above is of course not a precise analogue of the DEPENDS ON clause because the latter uses semantic information. To achieve exactly the same effect it is necessary to embed a special action in the syntax which would cause exit from the analyser on a count condition.

Let us now consider how the attribute names are linked with the values in the record. In COBOL this is a straightforward task since each attribute name delineates a fixed data area within the record area and this is the same for all records. The availability of alternative productions in a syntax scheme precludes the use of a fixed storage map. One may however analyse the data as it is read in and build up a separate storage map for each record. This was done for the more frequently accessed attributes in the integrated circuit records. For the other attributes, the storage map was in effect re-created by syntax analysis every time an access was required. The speed of this process was enhanced by holding tags at intervals throughout the record and by

Downloaded from https://academic.oup.com/comjnl/article/12/3/225/363357 by guest on 19 April 2024

performing a full analysis only between the markers which spanned the requisite value. With a data description as general as that provided by a CFPSG, the possibility of introducing ambiguities must always be guarded against. The program SID was of assistance here and as a by-product it produced a mapping of the data that was optimal in terms of the analysis speed.

Attribute grouping is not the only data activity for which the syntax approach proved helpful. A CFPSG also assisted in the formation of a data value for storing in a record such as a string or number. Many attributes have values which are directly represented by either one of these types.

e.g. *stock level* = 10 *or place of manufacture is France.*

Nevertheless there are many cases where the value is not so simple. Therefore, to avoid imposing rigid standardisation on the user or having to employ a rather hit or miss technique of string matching when searching, it is highly desirable to transform an external representation of a value into a canonical stored form. Where values may be represented by a single word the conversion of synonyms into a canonical form can be achieved using the vocabulary alone. For more complex cases it is necessary to use a syntax description with appropriate actions to achieve the conversion; such a syntax will be called a value syntax.

This technique was used in several places in the integrated circuit data base. One such conversion has already been indicated in Fig. 1. A more complex example is provided by the syntax necessary to describe the logical function performed by an integrated circuit. One such function is a gate, and in its simplest form, the number of inputs to a gate and its logical function have to be specified. A typical example is

3 *input nand.* (1)

However, inside one integrated circuit the inputs to one gate may be outputs from other gates which have to be specified. If the three inputs to a *nand* gate are the outputs from a *three input or* gate and two *two input or* gates, the specification of the function is

(3 *input or*, 2 *input or*, 2 *input or*) *nand.* (2)

Since there can be any number of gates in this sequence, the syntax must be recursive. A value string is made up by associating symbols with various parts of the syntax, so that when a function has been syntax analysed a sequence of symbols is left on the action stack, each distinct function leaving a different sequence of symbols. Another action then forms a single string from the symbols on the action stack.

Part of the syntax for function is

⟨*function*⟩ ::= 'g' ⟨*gate*⟩ | 'f' ⟨*flipflop*⟩ | . . .
⟨*gate*⟩ ::= ⟨*inpart*⟩ ⟨*logicword*⟩
⟨*logicword*⟩ ::= *and* 'a' | *or* 'o' | *nand* 'n' | *nor* 'r'
⟨*inpart*⟩ ::= ⟨*integer*⟩ *input* ('o' ⟨*gatelist*⟩) 'c'
⟨*gatelist*⟩ ::= ⟨*gate*⟩ | ⟨*gate*⟩ , 'y' ⟨*gatelist*⟩

The examples (1) and (2) will yield the strings *g3n* and *go3oy2oy2ocn*.

The use of syntax techniques to produce a canonical representation of a data item implies that a converse mechanism is required for printing values. Here again the analyser may be used with its input taken as a stream of single characters from the canonical representation and its output going directly to the printing device. Yet another syntax map is required for this process though it obviously bears a strong resemblance to the corresponding input syntax. For example: the syntax map used for printing functions is almost identical with the one used for reading in functions, but with the terminal symbols and the actions or meanings reversed. The analysis is followed by the call of a subroutine which prints out the items on the action stack.

It is also possible for a value to take the form of a reference pointer to another record. Reference values are not explicitly catered for in COBOL, nor were they used in the integrated circuits scheme, nevertheless it is interesting to see how they fit into a syntax scheme. Consider the standard example of a family record in the following simplified form:

⟨*personlist*⟩ ::= ⟨*person*⟩ ⟨*personlist*⟩ | ⟨*person*⟩
⟨*person*⟩ ::= ⟨*name*⟩ ⟨*age*⟩ ⟨*father*⟩ ⟨*mother*⟩
⟨*father*⟩ ::= ⟨*person*⟩
⟨*mother*⟩ ::= ⟨*person*⟩
⟨*name*⟩ ::= *identifier*
⟨*age*⟩ ::= *integer*

One interpretation of this syntax could be a world genealogy listing back to Adam and Eve within the single record (like the families of integrated circuits above). Alternatively we may regard ⟨*father*⟩ and ⟨*mother*⟩ both as references of type person and then each record will be limited to four entries. In large scale data retrieval schemes there will be several distinct files with different syntax descriptions. There is no reason in principle why references should not be allowed between such files by cross linking the distinct syntaxes by use of reference classnames.

So far the syntax which describes attribute grouping and the syntax used to form values have been considered separately. However the same formal method of specifying the syntax is used in each case so it is quite feasable, and in certain cases profitable, to merge the two distinct syntaxes into one comprehensive syntax. The possession of the unified syntax then allows a more flexible approach to attribute grouping since appropriate values may be resolved into finer substructure at search time. Consider the case of the logical function value for integrated circuits described above. It is possible to pass the canonical representation, treated as a character stream, through a further stage of analysis at search time. This re-constructed sub-structure may then be examined for any of its component values such as a particular logicword on a given level of the gate tree.

Summarising, we have found that the technique of syntax description can be consistently applied to the input, accessing and output of data. This technique has the advantage that the design of the program as a whole is more clearly separated from the detailed form of the data in a particular application, and a single analysis procedure is all that is required to achieve this end.

Downloaded from https://academic.oup.com/comjnl/article/12/3/225/363357 by guest on 19 April 2024

7. The retrieval language

The Command Language for Interrogating Computers (CLIC) which is described in this section is a simple language for data retrieval. The formal syntax is given in Appendix 3 and full details are given in a users' manual (Fox and Edwards 1968a, available on application from the authors). However a more discursive approach is adopted here in order to bring out the design features. The examples are all drawn from the RREAC implementation and reflect the vocabulary appropriate to integrated circuits. It must therefore be explicitly stated that CLIC itself is completely general and could be applied to searching any type of data simply by changing the search items and selection criteria.

Basic CLIC

The basic design aim was to provide a simple way of asking for information about items which satisfy a given set of selection criteria. The following example shows a typical CLIC command which incorporates these basic features.

print names of all circuits with cost ⟨2/0/0.

This command is built up from three phrases, viz.

request phrase	**subject phrase**	**choice phrase**
print names of	*all circuits*	*with cost* ⟨2/0/0.

These three phrases are the basic building blocks of any sentence in CLIC. The **subject phrase,** as the name implies, indicates which circuits are to be considered when making a choice so naturally it must always be present in a command. The **choice phrase** considers the circuits given by the **subject phrase** and selects from amongst them any which satisfy the listed criteria. Finally, the **request phrase** prints out any desired property of the chosen circuits.

The **choice phrase** contains **choice terms** which are typically in the form of the name of a property, a relation operator and a value, as in:

place of manufacture equal usa
cost ⟨5/0/0
propagation delay = unknown

(Note the use of the word *unknown* which is a perfectly acceptable value.) Alternatively, if the required property has a numerical value, then **choice terms** like

greatest stock level
smallest propagation delay

are available. Usually selection will be on the basis of more than one property. This can be expressed by linking up the **choice terms** into **choice expressions** using the connectives *and* or *or*, e.g.:

propagation delay ⟨10 *or greatest value of maximum counting rate*

Following the usual convention the *and* operator binds more tightly than the *or* operator, though any order of evaluation may be achieved by using round brackets as in

place of manufacture equal usa and (*maximum counting rate*⟩ 1000 *or propagation delay* ⟨5)

The **request phrase** may also be elaborated to enquire about several properties at once by linking the desired properties with the word *also*, for example:

print name also stock level of all circuits with propagation delay ⟨20.

Frequently it is useful to be able to ask for details about specific circuits whose names are known. To cater for this requirement an alternative **subject phrase** of the form

circuits **name, name** . . .

is allowed. It is also permissible to drop the **choice phrase** to give queries of the type

*print stock level of circuit ferranti micronor*2 *zss*83*a.*

If any of the data items being printed out have been annotated then the user will be informed of the reference numbers of the relevant footnotes. The contents of the notes can then be examined using a **request phrase** like the one in the command

print notes 1, 3, 5 *of circuit ferranti micronor*2 *zss*83*a.*

On-line aids

It is expected that the enquirer will approach his selection by a series of commands. Initially the conditions will cast the net widely and each subsequent command will apply successively more stringent conditions. To avoid the tedious repetition of previous selection criteria with each new question an alternative **subject phrase,**

above circuits

is provided. This has the effect of making the selection process given in the command apply only to the list of circuits derived in the previous sentence, e.g.

print total number of all circuits with place of manufacture equal usa.
print names of above circuits with smallest cost.

This facility may be generalised to allow the user to refer back to any stage in the selection process by allowing one to incorporate a **remember phrase** in any command between the **request phrase** and the **subject phrase.** The **remember phrase** has the simple form

also dub **name**

and has the effect of attaching a name devised by the user to the list of circuits chosen in that command, e.g.

request	**remember**	**subject**	**choice**
print names of	*also dub usmade*	*all circuits*	*with place of manufacture equal usa.*

Thereafter the name which has been given to a list of circuits may be used in a subsequent **subject phrase** instead of *above* or *all*, e.g.:

print names of usmade circuits with stock level⟩ 20.

The *above circuit* facility is in fact provided by automatically dubbing the name *above* on to the list of circuits generated by each command, whether or not a

Downloaded from https://academic.oup.com/comjnl/article/12/3/225/363357 by guest on 19 April 2024

remember phrase is used. Thereafter the word *above* appears to the next subject phrase just like any other dubbed name.

The following sequence of commands illustrates typical uses of a **remember phrase**:

*Q*1 *print names of also dub usmade all circuits with place of manufacture equal usa.*
*Q*2 *print names of also dub ukmade all circuits with place of manufacture equal uk.*
*Q*3 *print smallest price of usmade circuits.*

Here let us assume that the printed result is 1/0/0. The user might now continue with

*Q*4 *dub cheap ukmade circuits with price* ⟨1/0/0.
*Q*5 *print names of cheap circuits with can type equal flatpack.*

In order to make comparisons between the costs of the two groups of circuits it was necessary to retype the answer to one question (*Q*3) back into a later one (*Q*4) even though the value itself was uninteresting. To avoid this situation the language allows the value in any **choice term** to be replaced by a command to provide that value, e.g.:

print names of ukmade circuits with price ⟨smallest price of usmade circuits and can type equal flatpack.

In general the property used in the **choice phrase** and the property demanded of the inner sentence will be the same. To avoid needless repetition the use of the pronoun *that* is permitted in the inner sentence, e.g.:

. . . propagation delay ⟨that of . . .

The interpretive system used to implement CLIC holds both the vocabulary and the syntax as data to the analyser. Thus either may be modified on-line under the actions generated by specific commands.

Alteration of the vocabulary to allow the user to define synonyms is trivial. A typical synonym definition is illustrated by the command

define synonyms
ti for texasinstruments,
mfr for manufacturer,
hot for temperature.

which may appear at any point in a CLIC dialogue. Provision for the on-line modification of the syntax rules themselves was more complex. Consider the example,

define **choice expression**
temperature range at least **x** *to* **y**
where **x** *is* **tempval**, **y** *is* **tempval**

to mean

operating temperature minimum ⟨**x** *and*
operating temperature maximum⟩ **y**.

Here a new alternative production for **choice expression** is introduced. First a new action is constructed from the actions developed by the phrase expressed in the old syntax. Then the new production and action are merged into the old syntax map. This facility is powerful but has operational difficulties which prevented it from being generally released.

The immediate difficulty is that the user needs to know the identifiers for the class names (in second example they are **choice expression** and **tempval**). If the syntax has been optimised using SID (Foster, 1968), as it was in the RREAC implementation, there will probably be few distinct and meaningful classnames left. Indeed it might be necessary in some cases to return to the original syntax in order to express a definition readily. This extended syntax could then be re-optimised before use; this process would have the advantage of revealing ambiguities introduced into the new syntax. Even so, in a system which was released to the general user, it would be preferable to allow additional definitions to be made for only a few well-chosen class names. The additional safety precaution of providing each user with his distinct copy of the syntax map would also have to be made.

The interrogation language described above has proved easy to learn and to use. In addition, for the designers of the system as a whole, the syntax approach enabled the structure of CLIC to be enhanced with very little difficulty in the course of its development.

8. Implementation

It has already been remarked that the language is appropriate to a data base which possesses some structure. Nevertheless within this loosely defined requirement, CLIC could be applied to a variety of data bases since the exact nature of the actions is an implementation choice. The RREAC scheme uses the data base in the form described in Section 6. Here the actions invoke procedures of two main groups. First there are the file access procedures which search appropriate files (either serial or inverse) in the data base. In addition there are the procedures which operate on the results of the file access procedures, e.g. set union or intersection on lists of circuits, count of number of items in a list, or print-out of values in a list. All these procedures and the file structures they act on, are detailed in the paper 'The Implementation of a Data Retrieval Scheme on RREAC using Syntax Techniques' (Fox and Edwards, 1968b). The manner in which CLIC phrases are translated into calls of these procedures bears a strong resemblance to that used for implementing general purpose computer languages by means of a syntax specified translator system. Since this is a well understood task it does not seem necessary to present here precise details of the actions and the way in which they are incorporated into the CLIC syntax.

9. Discussion

The foregoing account describes the present state of the RRE retrieval scheme. The implementation has used a syntax driven interpreter throughout and this has conferred substantial advantages over *ad hoc* methods. For the syntax description is easy to understand, and the actions are distinct and have well defined side effects. Thus it has been possible to use a data base which is non-trivial in scale and complex in structure. Furthermore it has aided the design and implementation of the retrieval language which is powerful and yet easy to assimilate.

Downloaded from https://academic.oup.com/comjnl/article/12/3/225/363357 by guest on 19 April 2024

Two developments are at present under consideration for the CLIC system. The first is greatly to extend the range of procedures which may be applied to the retrieved data. In the present scheme these are limited to calculations of total, maximum or minimum values, but for many applications it would be useful to be able to apply a full range of statistical calculations. The use of infixed operators to allow simple arithmetical operations on data values can also be readily incorporated. The fact that these operations must be obeyed interpretively would not perceptibly alter the speed of response since 'programs' of the CLIC type are so trivially short. The second development is to allow certain names of subjects or criteria to be generic and to make the analyser perform the necessary treeing down to the level of the constituent terminal items.

Finally we would like to suggest that syntax specified languages should be considered for use in driving a wide range of application packages. Within these special 'problem' areas no single all-embracing language can be designed. The need is to be able to develop special one-off languages in as painless a manner as possible and to avoid the irksome restrictions and clumsy input formats of some present-day commercial packages. Translator writing systems appear to provide the mechanism for achieving this aim.

Appendix 1

Algorithm: A predictive analyser for syntax with embedded actions

consider the given initial classname.
replace that classname by a set of predictive stacks where each stack corresponds to an alternative expansion prescribed by the relevant syntax rule.
associate an action stack with each of the prediction stacks.

SCAN INPUT
read in the next symbol from the input sentence
SCAN STACKS
examine the first item in the first of the prediction stacks
if the first item is a terminal symbol
then

TERMINAL MATCH
begin
if it matches the input symbol
then remove that item from the stack and move the remainder of that stack together with its associated action stack from the set of prediction stacks into a set of accepted stacks
otherwise discard that prediction stack and also its associated action stack
end
otherwise if the first item is a classname
then replace it with the set of alternatives prescribed by the syntax. In each case the remainder of the original prediction stack is associated with each of the new alternatives. (In our system this remainder was a common sublist and not a fresh copy.)
otherwise if the first item is the name of an action
then transfer the actions corresponding to that name from that prediction stack into the associated action stack.

The examination of the prediction stack is repeated, from the point marked SCAN STACKS, until all the prediction stacks are either eliminated or transferred to the accepted stacks.

If there are no prediction parts left in the accepted stacks then the process is complete and the action stacks, if any, are transferred to the interpreter. Otherwise, the accepted stacks are treated as prediction stacks and the whole process repeated from the point marked SCAN INPUT.

Appendix 2

Algorithm: Incorporation of vocabulary in analyser

Two sections of the original analyser are expanded.

1. The line labelled SCAN INPUT now becomes:

SCAN INPUT
use an input routine to read the next word from the input sentence
search the vocabulary for that word
if present
then replace the incoming word with the list of terminal classes to which it may belong. In each case any possible meaning is associated with the relevant terminal symbol
otherwise examine incoming word and place in appropriate terminal class of identifier, real or integer with itself as meaning

2. The section labelled TERMINAL MATCH is changed to be:

TERMINAL MATCH
begin
take the next possible terminal symbol and meaning from the list given by the vocabulary
if this matches the first item in the prediction stack
then remove that item from the stack and move the remainder of that stack together with its associated action stack from the set of prediction stacks into a set of accepted stacks
otherwise
begin
discard that possible terminal symbol and meaning.
if further interpretation of the incoming word is possible
then return to TERMINAL MATCH
otherwise discard that prediction stack and also its associated action stack
end
end

Appendix 3

CLIC Syntax

The Syntax of CLIC is given here in Backus Normal Form, with the following additional conventions:

Downloaded from https://academic.oup.com/comjnl/article/12/3/225/363357 by guest on 19 April 2024

(1) a terminal symbol in quotes means that any identifier is acceptable at that place;
(2) the symbol $ at the end of a word (e.g. circuit$) means that either singular or plural form of the word is allowed;
(3) ∅ as an alternative for a syntactic class means that class need not be present.

⟨*command*⟩ ::= ⟨*retrieval command*⟩. | ⟨*other command*⟩.
⟨*retrieval command*⟩ ::= ⟨*request*⟩ ⟨*remember*⟩ ⟨*subject*⟩ ⟨*choice*⟩
⟨*subject*⟩ ::= *all circuits* | *above circuits* | '*dubname*' *circuits* | *circuits* ⟨*namelist*⟩
⟨*namelist*⟩ ::= ⟨*circname*⟩ | ⟨*circname*⟩, ⟨*namelist*⟩
⟨*circname*⟩ ::= '*manufacturer*' '*series no*' '*order no*'
⟨*choice*⟩ ::= *with* ⟨*choice expression*⟩ | ∅
⟨*choice expression*⟩ ::= ⟨*ce*1⟩ | ⟨*ce*1⟩ *or* ⟨*choice expression*⟩
⟨*ce*1⟩ ::= ⟨*choice term*⟩ | ⟨*choice term*⟩ *and* ⟨*ce*1⟩
⟨*choice term*⟩ ::= (⟨*choice expression*⟩) | ⟨*word property*⟩ ⟨*EO*⟩ ⟨*value*⟩ | ⟨*number property*⟩ ⟨*RO*⟩ ⟨*value*⟩ | ⟨*qualifier*⟩ ⟨*number property*⟩
⟨*word property*⟩ ::= *name* | *can type* | *logical function* | . . . etc.
⟨*number property*⟩ ::= *cost* | *stock level* | *propagation delay* | . . . etc.
⟨*EO*⟩ ::= *equal* | *not equal*
⟨*RO*⟩ ::= = | ≠ | < | ⩽ | > | ⩾
⟨*qualifier*⟩ ::= *smallest* | *smallest value of* | *greatest* | *greatest value of*
⟨*value*⟩ ::= ⟨*embedded command*⟩ | ⟨*value syntax*⟩ | *unknown*
⟨*value syntax*⟩ depends on the property being considered
e.g. ⟨*value syntax for cost*⟩ ::= *integer* / *integer* / *integer* | *integer* / *integer*
⟨*request*⟩ ::= *print* ⟨*pplist*⟩ *of* | *select* | ∅
⟨*pplist*⟩ ::= ⟨*pp*⟩ | ⟨*pp*⟩ *also* ⟨*pplist*⟩
⟨*pp*⟩ ::= *total number* | ⟨*word property*⟩ | ⟨*number property*⟩ | ⟨*qualifier*⟩ ⟨*number property*⟩ | *that*
⟨*remember*⟩ ::= *also dub* '*dubname*' | *dub* '*dubname*' | ∅
⟨*embedded command*⟩ ::= ⟨*request*1⟩ ⟨*remember*⟩ ⟨*subject*⟩ ⟨*choice*1⟩
⟨*request*1⟩ ::= ⟨*pp*⟩ *of* | *print* ⟨*pp*⟩ *of*
⟨*choice*1⟩ ::= *with* ⟨*choice term*⟩ | ∅
⟨*other command*⟩ ::= ⟨*forget*⟩ | ⟨*synonyms*⟩ | ⟨*directive*⟩
⟨*forget*⟩ ::= *forget* ⟨*dubnamelist*⟩
⟨*dubnamelist*⟩ ::= '*dubname*' | '*dubname*', ⟨*dubnamelist*⟩
⟨*synonyms*⟩ ::= *define synonyms* ⟨*synonym list*⟩
⟨*synonym list*⟩ ::= '*new*' *for* '*old*' | '*new*' *for* '*old*', ⟨*synonym list*⟩
⟨*directive*⟩ various commands to select input and output channels, and anything else required by a particular implementation.

References

FELDMAN, J., and GRIES, D. (1968). Translator Writing Systems, *Comm. ACM*, Vol. 11, No. 2, p. 77.
FOSTER, J. M. (1967). Interrogation Languages, *Machine Intelligence* 1, p. 267. Edinburgh and London: Oliver and Boyd.
FOSTER, J. M. (1968). A Syntax Improving Program, *Computer Journal*, Vol. 11, No. 1, p. 31.
FOX, A. J., and EDWARDS, P. W. (1968a). The Selection of Integrated Circuits using RREAC—Language Manual. Ministry of Technology unpublished work.
FOX, A. J., and EDWARDS, P. W. (1968b). The Implementation of a Data Retrieval Scheme on RREAC using Syntax Techniques. Ministry of Technology unpublished work.
LANDIN, P. J. (1964). The Mechanical Evaluation of Expressions, *Computer Journal*, Vol. 6, No. 4, p. 308.

Book Review

Advanced Linear-Programming Computing Techniques, by William Orchard-Hays, 1968; 355 pages. (McGraw-Hill, £5 17s. 0d.)

This is an ideal reference book for those involved in using, writing, or amending programs for linear programming. The application theory, and the detailed interpretation of the results are not examined. One objective of the book is that it will establish a notation and nomenclature to promote easier communication. It is not considered that the notation satisfies this aim.

The introductory chapters define the problem and discuss the simplex methods of solution. Although the book is claimed not to provide a mathematical background, it should be pointed out that it is largely concerned with mathematics and suggested algorithms for the writing of computer programs, and it is therefore advisable that readers are well versed in algebraic techniques. An excellent illustrative example of a 'widget' manufacturer is included, this continues through the book, which shows how these programs can be applied. For those assisting management in the formulation of company models, these sections will be of considerable assistance.

Another example, for the manual solving of simultaneous equations is rather unfortunate in that it makes an easy problem difficult. This is, however, a minor criticism.

The procedures for the ranging of coefficients in the modification, extension, and combination of company models are of particular interest. In certain sections mnemonics come thick and fast, PARROW, PARCOL, PARRHS, etc., as detailed computer programs are investigated. The development of decomposition techniques, including the Dantzig-Wolfe Algorithm, are well documented. The Appendices on Elementary Transformation and the Mathematical Programming System are excellent and there is also a useful index.

In conclusion, the author comments that the techniques developed over the last ten years have automated Linear Programming computational runs; but with the increased power of program systems and algorithms, it is likely that it will again be necessary for the analyst to direct the procedures.

This book can be recommended both for Scientific Computer Programmers and for Operational Research personnel who are involved in the application of linear programming.

G. P. D. MORRIS (Coventry)

Downloaded from https://academic.oup.com/comjnl/article/12/3/225/363357 by guest on 19 April 2024