QUAESTIONES INFORMATICAE

Vol. 1 No. 2

September, 1979

Quaestiones Informaticae

An official publication of the Computer Society of South Africa
'n Amptelike tydskrif van die Rekenaarvereniging van Suid-Afrika

Editors: Dr. D. S. Henderson,
Vice Chancellor, Rhodes University, Grahamstown, 6140, South Africa.
Prof. M. H. Williams,
Department of Computer Science and Applied Maths,
Rhodes University, Grahamstown, 6140, South Africa.

Subscriptions

Annual subscriptions are as follows:

	SA	US	UK
Individuals	R2	$3	£1.50
Institutions	R4	$6	£3.00

Quaestiones Informaticae is prepared for publication by SYSTEMS PUBLISHERS (PTY) LTD

for the Computer Society of South Africa.

The Sequence Abstraction in the Implementation of Emily

D.C Currin, J M. Bishop and Y L. Varol*

Computer Science Division,
University of the Witwatersrand,
Johannesburg,
South Africa.

Abstract

Emily is a general purpose computer aided instruction language with a novel implementation At the user s level, Emily consists of two languages an instruction language, which is used by authors to prepare lessons and a command language for controlling access to lessons, registering users and other system functions
In implementation, such a system poses the problem of storing and controlling a wide variety of data types under program control and on backing store Both storage and control are handled by a data base manager whose basic unit is a sequence
This paper gives a brief description of the Emily languages and then concentrates on the sequence abstraction It is shown how this provides an effective interface between concepts in the high level implementation language (Pascal) and the details of the realization of many different data types in a single system Finally, the internal representation of sequences is discussed

1 Introduction

In 1977, a language-generator pair for automatic problem generation in computer aided instruction was proposed [4] and a pilot implementation undertaken This preliminary investigation revealed deficiencies in the language, which was based on functions and assignments, without any control structures Once these had been cleared up in the revised report of the language [2,3] the implementation was approached afresh This presented two problems Firstly, to be viable, Emily, as the system is now called, should be embedded in an environment which includes facilities for storing and retrieving problem prototypes (or lessons) and for registering and monitoring the use of these prototypes This necessitated the design of a command language to handle the interaction between a parent, system, a database manager, the users on terminals, and the generator-verifier
Secondly, there was the problem of ensuring a secure yet efficient production system A system such as Emily generates a large number of different data types, all highly structured and inter-connected It is desirable that these types be reflected by the implementation language and that the protection afforded by strict typing in a language such as Pascal be retained However, strict-typing means that procedures may only serve items of a particular type and that connections via pointers must be between specified types For example, data in an Emily lesson may vary from simple characters to vectors of numbers and strings and all of these need to be stored and manipulated on a stack while the lesson is in progress Following strict-typing, this leads to a proliferation of versions of each logical procedure and the indiscriminate use of type unions (or record variants) The naming conventions alone make this solution unwieldy
While this dilemma was posed chiefly in relation to the generator-verifier pair, which had already been partially implemented, it also has a bearing on the action of the database manager This should not be required to manage more than one or two types of objects in the transfer between main memory and backing store
The answer to the problem lies in abstracting the definition of the data types and structuring methods away from their representation in a language (albeit a high level one) For this purpose, the sequence abstraction is ideal Every data structure is regarded as an indefinite sequence of non-homogeneous terms, which may themselves be sequences The operations on sequences are few but adequate and security is achieved by means of a well-defined procedural interface Thus the actual representation of sequences is immaterial at the programming level and the design of the major Emily modules could proceed independently of the decisions as to which of the existing structured types (pointers, arrays, files) would be used
This paper has three aims Firstly, Emily and its capabilities are described, in order to establish the magnitude of the project being undertaken Secondly, it is shown how the sequence abstraction simplifies the design of the data structures required, and keeps these at a uniformly high level Thirdly, an implementation of sequences is described, to give an idea of the fundamental difference between the abstraction of sequences and their representation

2 The Emily Environment

2 1 Modules in the System

Figure 1 gives a diagrammatic view of the various modules in Emily The Controller and Interpreter are the two that will be most widely used for the execution of lessons The compiler will be invoked by authors of lessons All four main modules will use the Data base manager, the analysers and the General Purpose modules This level of interaction makes it essential for there to be a standard data type for communicating between the modules

*Y L Varol is now at the University of Southern Illinois at Carbondale, U S A

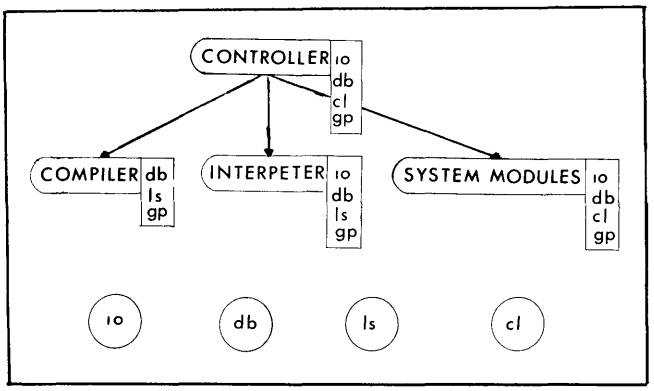

Figure 1 Modules of Emily
io = input/output modules
db = database manager
ls = lexical and syntax analyser
cl = command language analyser

2.2 The Instruction Language

The instruction language is used by authors to define *lessons* The author is able to write lessons which are adaptable The difficulty of the subject matter covered, as well as the speed at which it is covered, is under the control of the author
The lessons can be to a large extent generative This makes it possible to give a user extensive practice in sections with which he is unsure
The instruction language is easily readable, and the lessons are modular This enables authors to write lessons which can be easily read and modified by other authors
The lessons are transportable The language does not assume any particular hardware For implementation dependent features (such as the number of significant digits which can be retained) the report [2] describes the minimum requirements
The basis for generating an instance of an Emily lesson and for validating the replies, is the *function* Each function takes a single argument and returns a value of one of the basic types Because the argument may be a vector, the effective argument can be as sophisticated as necessary and may even include nested functions In addition, some functions may be qualified by specific attributes For example
REDUCE 2x + 3y – 1 SUBST [y, number]
will reduce the given polynomial to its canonical form and make the given substitution
A validation function has an argument which is matched against the users replies If this results in a match the function is set to boolean true (1), otherwise it has the value false (O)
ONEOF[metre , metres , m'] – will be true if the reply contains any one of the strings
Variables, parameters and expressions may be of type boolean, numeric, polynomial, string, picture or vector
Simple statements in Emily provide for the assignment of values to variables, the invocation of lessons, presentation of pictures and requests for replies The compound statements include the IF, REPEAT and FIRSTOF statements Declarations of parameters occur at the start of the lesson, and thereafter the control structures take over

2 3 The Command Language

Using the command language the user may perform actions such as using a lesson, creating, deleting or altering lessons or users Objects such as lessons are not restricted to a particular user but are available to certain divisions of the user community When objects are created they are registered under a particular division Some example of divisions are
science
science appliedmaths
science appliedmaths group1
Users are able to access objects which fall under their own division They are also able to use lessons which are registered under a less qualified division Privileged users are able to sub-divide a division up into further divisions
In order to do this, and other tasks such as creating lessons, a user requires certain attributes It is possible to register certain categories of user, such as student, author or supervisor, and to give them the classes of attributes that they will require
The command language has a facility for adding different systems to the basic system In our implementation we will have an I/O system (which is mandatory) a test system, and a database system

2.4 Examples

```
country   ELEMENT[France  Japan, Portugal, Italy, China]
          index n,
capital   ELEMENT[Paris, Tokyo, Lisbon  Rome  Peking]
          index n,
REPEAT
n ← range [1,5]
Ask concat[ What is the capital of , country index n]
If keyword capital index n
    then present  Well Done
    else present  Try another one ,
UNTIL keyword capital index n
```

(Notice that Emily does not distinguish between upper and lower case and an author is free to use them for emphasis as in the above example)
The firstof statement selects one of a number of statements for execution The statement corresponding to the first Boolean expression to yield the value 1 is executed An example of an Emily lesson to teach its own command language would be

```
commands ( create', delete', display', user      )
ASK '?'  {The prompt}
FIRSTOF
valid['create   oneof[ user   lesson ]]
      createcommand,
valid[ delete , oneof[ user ,  lesson ]]
      deletecommand,
otherwise
      Present concat[ Here are the available ,
           commands', LIST commands],
END
```

3 The sequence abstraction

3 1 Motivation

When designing the implementation of Emily [3] we found that there are a large number of data types to be stored and manipulated Some examples are identifiers, expressions, polynomials, vectors, statements lessons users, categories and divisions
All this data needs to be stored in a database Conceptually, the

data needs to be stored in a form which is easy to envisage and to manipulate Ideally the code for a lesson would correspond directly to the lesson syntax In order that a safe implementation may be obtained it is essential that the different data types be clearly distinguished, and that strict type checking be used, such as is available in Pascal

On the other hand it is preferable that the data stored in the database be treated independently of its type, and that it can be stored in fixed size chunks Otherwise different procedures would be required to store or fetch each of the data types, leading to an impossibly complex system

The sequence abstraction is a way of realizing these two seemingly conflicting objectives A *sequence* is an indefinite number of terms arranged in linear order The terms may store a unit of data such as a real number, or may themselves be other sequences

3 2 Defining Sequences

All data used in the system may be represented by sequences When compiling the source text the compiler will gradually build up a data structure of sequences Later, when a user wishes to use the lesson, the interpreter will take this structure and perform the required actions The following definition of an expression follows almost directly from its syntactic definition in the language report [2]

```
expression = SEQUENCE of (operator) (function) (constant)
(variable) (parameter) (expression)
operator = SYMBOLIC exp, mult, div, plus, minus, negate,
equal, noteq, andop, orop
etc
```

Instances of the types of terms enclosed in round brackets may appear in the sequence any number of times and in any order Sequences may be shared concurrently between processes servicing different users For example, the sequence constituting a lesson may be shared if more than one user is simultaneously using that lesson These sequences are termed *free sequences* On the other hand the particular values for variables in the lesson are only available to one process These are termed *private sequences*

3 3 Accessing Sequences

Each sequence is identified by a unique sequence number, and has a fixed type This type determines what the terms of the sequence may be For example, a sequence of type expression may only contain terms of type parameter, variable, expression, constant, operator and function Those terms which are in turn sequences are represented simply by the relevant sequence number

When accessing a sequence the process must specify what the type of the sequence will be In this way it is easy to detect if the process has encountered a corrupted sequence In this eventuality the database manager will be able to terminate the affected process and to give information regarding the last sequence successfully accessed by that process The implementor may then ascertain whether the sequence was really corrupted, or whether the process was at fault In our implementation we are providing a permanent database system to deal with such problems Using this system a privileged user may perform any of the operations on a sequence which could be performed by a process

Each instance of a (possibly shared) sequence has a window Using this window a process may examine a particular term, may advance the window forwards with get", or may 'skip ' a relative number of terms forwards or backwards A process may create and delete private sequences Terms may be inserted into or removed from a private sequence Once a private sequence has been set up the process may release it for general use This sequence will then be classified as free Other processes may open it, may examine the terms using get and skip, and when finished may close it again If a process wishes to alter a free sequence it must appropriate it This will change it back into a private sequence All other processes wishing to access the sequence will be put into a wait state until the sequence is released or deleted

3 4 An example

To conclude this section we will give an example of the use of this sequence abstraction Associated with each user is a number of pieces of information, for example, the user s identifier, his name and password, the attributes that he possesses, and the division to which he belongs Each user also has a record of the lessons which he has used Some of these may be lessons which he has temporarily suspended, and with which he will later continue Other lessons may have been completed A record is kept of how well the user performed This information is retained in a lesson history

```
user = SEQUENCE OF[useridentifier] [name] [password]
       [division] [attributes] (usage),
usage = RECORD lessonid, history
```

Those terms which are enclosed in square brackets will appear in the sequence in that order The terms which are enclosed in round brackets follow after the ordered terms, repeatedly, and in any order

The Pascal code needed to locate a suspended lesson and to invoke the interpreter would be as follows

```
thisuser classification  = usersequence
open (thisuser),
skip(thisuser,5), {usage is term 5}
found  = false,
while not (found or thisuser eos) do begin
   found  = compare(thisuser t.usage lessonid, equal, givenlesson),
   if not found then get (thisuser),
end
close (thisuser),
if found then begin
    find(alesson {with}, givenlesson {as an id and return its},
        lessonsequence)
    interpret (lessonsequence, thisuser t usage history)
end
else {not found}
```

Two points to note are

1 The additional "t fields in thisuser are a result of translating sequences into Pascal records (see section 4) This is in fact done automatically by a preprocessor to the Pascal compiler
2 Alesson is an open sequence of all lessons
3 Compare will work for any type of sequences in this case it is used for strings which are defined as sequences of characters

4 The internal representation of sequences

Bishop[1] described a method for implementing strings in Pascal using the sequence abstraction The approach taken here is essentially a generalization of this idea

Sequences are stored as a linked list of chunks Each chunk can store a fixed number of terms As the sequence gets longer more chunks are added into the sequence When a chunk is no longer used it is reclaimed by the database manager Each chunk is referred to by a pointer The database manager is free to load sequences into and out of main memory whenever it sees fit It thus operates a virtual storage system

The database manager maintains a table of all sequences which

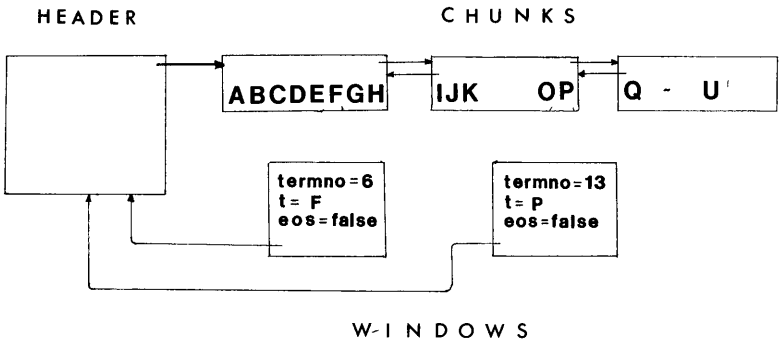

Figure 2 An Example of the relationship between sequence headers, chunks and windows

are currently in main memory Should a sequence which is not in memory be required, it will be fetched off backing store and might replace some of the sequences currently in memory The replacement strategy chosen is cyclic progression Chunks are defined as

```
TYPE chunk = RECORD
    next, previous  ↑chunk
    body  PACKED ARRAY [1  chunksize] OF term,
    present  SET OF 1  chunksize,
END,
```

where term is a variant record encompassing all possibilities Not all the terms in the chunk need be present When a term is present, its ordinal value is added to the set When terms are inserted into a sequence, the database manager uses up the unused terms within a chunk This may require that some of the terms in that chunk be shifted If all the terms in the chunk are used a new chunk will be added into the linked list

Conversely when a term is removed the associated ordinal number is removed from the set No shuffling of terms takes place If all the terms in the chunk are deleted the chunk is released When skipping along a sequence the database manager scans the list of chunks, subtracting off the number of terms present in each chunk, until it comes to the correct chunk It then scans all the terms within the chunk until coming to the required term

Each sequence has a header record, containing the classification of the sequence, whether it is free, private or appropriated and the number of the first chunk

```
TYPE header = RECORD
        first  ↑chunk,
        status  (private, free, appropriated)
        classification  sequence,
END,
```

For each process there is a window This also contains the classification of the sequence, defined as

```
TYPE sequence = RECORD
        itsnumber  natural,
        itstype  sequencetype,
        processcode  natural
END,
```

as well as an indication of the current term number, its value and an end of sequence flag

```
TYPE window = RECORD
        classification  sequence,
        t  term,
        termno  natural,
        eos  boolean,
END,
```

Figure 2 gives an example of the relationship between headers, chunks and windows

Whenever a private sequence is created or a free sequence is appropriated, the database manager will assign it a random process code This code will also be put in the window of the calling process Any process wishing to access the sequence thereafter will have to supply the correct process code

In the implementation of Emily, it is found that many of sequences will be strings Examples are the source text for a lesson, and all the string constants invariably found within the lessons Since the term character takes up far less space than the other types of terms, it seems that we should maintain two types of chunks The first type will store only character terms, whereas the other type will store any type of term In this way, we can store characters more efficiently The size of the two types of chunks will still be the same, so that they may be treated identically by the database manager

5 Conclusions

This paper gives a snapshot of the development of the implementation of Emily, somewhere at the end of the design phase It is highly likely that changes will occur in most aspects of the system, but the sequence abstraction ensures that these will be contained on one or other side of the interface, which will itself remain stable

The most important result of this work is the safety and reliability of an implementation based on the sequence abstraction, coupled with the realisation of a virtual storage system with only two sizes of data (chunks and headers) It is believed that these principles are generally applicable in the design of large software systems

Acknowledgements

The authors would like to acknowledge the assistance of Colleen Kelly in the pilot implementation of the system, and work of Ruth Chomse, Richard Searle and John Volsteedt whose questions and criticisms have moulded much of the work presented here

References

[1] BISHOP, J M (1979) Implementing Strings in Pascal, *Software – Practice and Experience,* to be published

[2] CURRIN, D C and BISHOP, J M (1979) *Emily Preliminary Report,* Applied Mathematics Internal Report, University of the Witwatersrand

[3] CURRIN, D C and BISHOP, J M (1979) *Emily Implementors Guide,* Applied Mathematics Internal Report, University of the Witwatersrand

[4] VAROL, Y L (1977) Automatic Problem Generation, Applied Mathematics Internal Report, University of the Witwatersrand

Notes for Contributors

The purpose of this Journal will be to publish original papers in any field of computing. Papers submitted may be research articles, review articles, exploratory articles or articles of general interest to readers of the Journal. The preferred languages of the Journal will be the congress languages of IFIP although papers in other languages will not be precluded.

Manuscripts should be in double-spaced typing on one side only of A 4 paper and submitted to Dr. D. S. Henderson or Prof. M. H. Williams at

Rhodes University
Grahamstown 6140
South Africa

Form of manuscript

Manuscripts should be in double-space typing on one side only of sheets of A4 size with wide margins. The original ribbon copy of the typed manuscript should be submitted. Authors should write concisely.

The first page should include the article title (which should be brief), the author's name, and the affiliation and address. Each paper must be accompanied by a summary of less than 200 words which will be printed immediately below the title at the beginning of the paper, together with an appropriate key word list and a list of relevant Computing Review categories.

Tables and figures

Illustrations and tables should not be included in the text, although the author should indicate the desired location of each in the printed text. Tables should be typed on separate sheets and should be numbered consecutively and titled.

Illustrations should also be supplied on separate sheets, and each should be clearly identified on the back in pencil with the Author's name and figure number. Original line drawings (not photoprints) should be submitted and should include all relevant details. Drawings, etc., should be about twice the final size required and lettering must be clear and "open" and sufficiently large to permit the necessary reduction of size in block-making.

Where photographs are submitted, glossy bromide prints are required. If words or numbers are to appear on a photograph, two prints should be sent, the lettering being clearly indicated on one print only. Computer programs or output should be given on clear original printouts and preferably not on lined paper so that they can be reproduced photographically.

Figure legends should be typed on a separate sheet and placed at the end of the manuscript.

Symbols

Mathematical and other symbols may be either handwritten or typewritten. Greek letters and unusual symbols should be identified in the margin. Distinction should be made between capital and lower case letters between the letter O and zero; between the letter l, the number one and prime; between K and kappa.

References

References should be listed at the end of the manuscript in alphabetical order of author's name, and cited in the text by number in square brackets. Journal references should be arranged thus:

1. ASHCROFT, E. and MANNA, Z. (1972). The Translation of 'GOTO' Programs to 'WHILE' Programs, in *Proceedings of IFIP Congress 71,* North-Holland, Amsterdam, 250-255.
2. BÖHM, C. and JACOPINI, G. (1966). Flow Diagrams, Turing Machines and Languages with only Two Formation Rules, *Comm. ACM,* **9,** 366-371.
3. GINSBURG, S. (1966). *Mathematical Theory of Context-free Languages,* McGraw Hill, New York.

Proofs and reprints

Galley proofs will be sent to the author to ensure that the papers have been correctly set up in type and not for the addition of new material or amendment of texts. Excessive alterations may have to be disallowed or the cost charged against the author. Corrected galley proofs, together with the original typescript, must be returned to the editor within three days to minimize the risk of the author's contribution having to be held over to a later issue.

Fifty reprints of each article will be supplied free of charge. Additional copies may be purchased on a reprint order form which will accompany the proofs.

Only original papers will be accepted, and copyright in published papers will be vested in the publisher.

Letters

A section of "Letters to the Editor" (each limited to about 500 words) will provide a forum for discussion of recent problems.

Hierdie notas is ook in Afrikaans verkrygbaar.

Questiones Informaticae

Partial proceedings of the first South African Computer Symposium on Research in Theory, Software, Hardware, organised by The Research Symposium Organising Committee of The Computer Society of South Africa. 4 & 5 September 1979, Pretoria.

Contents/Inhoud