QI QUAESTIONES INFORMATICAE

Vol. 3 No. 3 August, 1985

An official publication of the Computer Society of South Africa and of the South African Institute of Computer Scientists

'n Amptelike tydskrif van die Rekenaarvereniging van Suid-Afrika en van die Suid-Afrikaanse Instituut van Rekenaarwetenskaplikes

QUAESTIONES INFORMATICAE

An official publication of the Computer Society of South Africa and of the South African Institute of Computer Scientists

'n Amptelike tydskrif van die Rekenaarvereniging van Suid-Afrika en van die Suid-Afrikaanse Instituut van Rekenaarwetenskaplikes

Editor: Prof G. Wiechers
INFOPLAN
Private Bag 3002
Monument Park 0105

Editorial Advisory Board

Subscriptions

Annual subscriptions are as follows:

	SA	US	UK
Individuals	R10	$7	£5
Institutions	R15	$14	£10

Circulation and Production Manager

MR. C. S. M. Mueller
Department of Computer Science
University of the Witwatersrand
1 Jan Smuts Avenue
Johannesburg 2001

Quaestiones Informaticae is prepared by the Computer Centre and Central Graphics Unit, and printed by the Central Printing Unit, all of the University of the Witwatersrand, for the Computer Society of South Africa and the South African Institute of Computer Scientists.

Editorial Note

I am happy to present the current issue of Quaestiones Informaticae. As indicated in the last issue, Vol.3 No.2, it was necessary to look for a new method to produce the journal. Thanks to the efforts of Prof. Judy Bishop we have found a way to continue at an acceptable cost, and the result is in your hands. Wits University's Computer Centre will do the 'typesetting' using a laser printer. Judy will use Computer Science students to do the proof reading, and Wits will also do the printing. On behalf of myself, the Computer Society of SA and the SA Institute of Computer Scientists, I want to express my appreciation for all their efforts.

Conrad Mueller, of Wits' Computer Science department has agreed to become the circulation manager for QI. He will handle the business side of our publication. A hearty welcome to him!

G. WIECHERS
Editor.

STOP PRESS

QI is now on the official supplementary list of journals considered by the National Department of Education for subsidy purposes.

Quo Vadis, Computer Science?

K.J.MACGREGOR
Computer Science Department,
University of Cape Town,
Rondebosch,
7700

INTRODUCTION

The discipline of Computer Science has been established for almost twenty years. In SOUTH AFRICA, Computer Science Departments have been in existence for fifteen years, and producing graduates for twelve, however, the path ahead for Computer Science is really no clearer than it was in the early years. The different situation that exists in South Africa, and the pressure being exerted by the computing community, continues to cause doubts to be raised as to the relevance of Computer Science to the needs of the South African marketplace. This, in itself, is not of major concern if it is believed that the local computing community will mature into requiring the specialist knowledge for which a Computer Science course equips its graduates, however, this has not happened as yet and there is some doubt as to whether it will ever happen. As a result, perhaps, the time has come to consider whether Computer Science as a pure discipline, such as information processing should be taught as the norm in South African Universities with Computer Science relegated to a minor role.

This paper attempts to trace the history of computer science, describe the current views on the discipline, mention the pressures currently existing in South Africa and express some personal views on the direction Computer Science curricula should take in future.[1]

THE HISTORY OF COMPUTER SCIENCE

The formal start to Computer Science teaching began with the publication of the curriculum '68 [1] by the curriculum committee on computer Science. This curriculum established the status quo that existed at that time, by specifying a collection of modules which, when assembled, would constitute a core curriculum for Computer Science degrees. The curriculum did not specify any objectives for the courses and did not attempt to indicate any relevance of the course to practical computing and most certainly not to commercial data processing. The fifties and sixties were the eras of scientific computing and attempts at formalising computing concepts. Success had been achieved in the production of a specification of Algol 60 and more success was to be realised with the specification of Algol 68, or so it was hoped. In addition it was generally accepted, in academic environments, that all the innovations in the hardware of computers had been necessitated by the

---

[1] Presented as the Guest Lecture at the AGM of the Institute of Computer Scientists, October 1984.

demands of numerical computation and increasing computer power. The role of commercial computing was considered to be that of a group of practitioners whose needs could easily be satisfied by the equipment being produced and who do not have any special requirements.

The next step forward was taken in 1978 by the publication of the updated curriculum on Computer Science '78 [2]. This curriculum adopted a far more realistic view of Computer science and, in particular, the production of software. The seventies was the decade of the software crisis. The realism that software was difficult and expensive to produce, and even more difficult to achieve any kind of reliability. It was the decade of the introduction of program verification, structured programming, and the design of programs using predicate logic in pre- and post- conditions. This formalism in the specification also vested itself in the specification of an objective for the curriculum. The '78 curriculum stated that
"Computer science majors should:

1. be able to write programs in a reasonable amount of time that work correctly, are well documented and are readable;

2. be able to determine whether they have written a program that is reasonably written and is well organised;

3. know what general types of problems are amenable to computer solution, and the various tools necessary for solving such problems;

4. be able to access the implications of work performed either as an individual or as a member of a team;

5. understand basic computer architectures;

6. be prepared to pursue in-depth training in one or more application areas or further education in computer science".

These objectives have continued to guide Computer Science teaching in America in the eighties. The teaching of formal approaches to software design and implementation is even more in evidence than it was in the seventies. The adoption of this rigorous approach to computing ties in with the belief that certain basic theorems for software design have emerged.

THE NEEDS OF AMERICAN INDUSTRY

The teaching of Computer Science and the '78 curriculum match the need of the American society. The computer manufacturing and software industry has a voracious appetite for graduates particularly at a post graduate level. At the "SNOWBIRD" [3] discussions in 1978 it was stated that extremely few of the some 250 PhD. graduates produced in the U.S.A. per annum were remaining in the academic sector but most were going to commerce. One company, Xerox, admitted to hiring some 10 doctoral graduates per annum for at least 7 years and they are probably relatively modest hirers compared to IBM.and AT&T. Such is the demand from the smaller software houses that any suitably qualified Computer Science graduate can obtain a well paid job suited to his qualifications. At a personal level, I was conversing with a manager from an IBM research plant recently established in Toronto. This plant has responsibility for language design for all machines between the PC and the 3800. To perform its task it requires a staff of some 300 Computer Scientists.

After six months it had obtained 50 staff. It is not surprising that the law of supply and demand is resulting in the Computer Science curriculum

in the U.S.A. being tailored to this systems software and computer manufacture research environment.

THE NEEDS OF DATA PROCESSING IN THE USA

As Computer Science has its approved curriculum, so the data processing profession has produced its own curriculum. Information systems curricula were produced in 1972 [4] and 1973 [5] and subsequently updated in 1982 [6]. The 1982 curriculum was based on three curriculum proposals: The IEEE draft report on software engineering [7], the IFIP TC3 committees proposals for information system designers [8] and the DPMA model curriculum [9]. This '82 curriculum described the differences between Computer Science and Information Systems as:

1. The IS curriculum teaches information system concepts and processes within two contexts, organisation functions and management knowledge and technical systems knowledge, whereas computer science tends to be taught within an environment of mathematics, algorithms, and engineering technology.

2. The IS graduate is expected to work within the environment of an organisation and to interact with both organisational functions and computer technology. The computer science graduate has less interaction with organisational functions and more interaction with hardware and software technology.

3. In technical expertise the IS curriculum places substantial emphasis on the ability to develop an information system structure for an organisation and to design and implement applications. There is less emphasis on depth skills in hardware and software design. The Computer Science graduate typically has less exposure to information requirements analysis and organisational considerations but obtains greater expertise in algorithm development, programming, and system software and hardware.

The curriculum report further goes on to state that in general the entry positions for information systems graduates are:

1. Systems Analyst (entry level)

2. Application programmer or Programmer Analyst

3. Information systems specialist covering areas as information systems planning, administration and resource management.

From the background of this curriculum particularly the endorsement by the DPMA it is obvious that these graduates are orientated to the data processing profession and not the computer and software manufacturers.

THE HISTORY OF COMPUTER SCIENCE IN SOUTH AFRICA

Computer Science started in South Africa in the late 1960's early 1970's. The curricula at the various universities was stimulated initially by guest lecturers brought to the country with sponsorship provided by IBM. The curricula followed the philosophy of the curriculum '68. In 1980 the Computer Science Lecturers Association held a discussion to investigate the local curricula in the light of the objectives of curriculum '78. A core curriculum to be followed by all the local universities was developed. This curriculum was, by nature of the diversity of the universities, vague. In addition as no agreement could be reached on the prerequisites of any course, particularly those relating to

non-Computer Science subjects, such as mathematics, statistics, accounting, etc., the environment in which individual components were taught differ and as such the courses could, to all intents and purposes, be different courses although named the same. As no follow-up to this curriculum discussion has taken place at any subsequent CSLA meetings it is doubtful whether any universities actually follow any more than the broad outline of this curriculum. However, most Computer Science Departments still adhere to the concepts of the curriculum '78.

THE NEEDS OF INDUSTRY IN SOUTH AFRICA

Discussions have taken place regularly between industry and the Computer Science Departments in South Africa, over the relevance of Computer Science curricula. The most notable of these conferences were the Jan Smuts Conference [9] of 1973 and the Rosebank Conference of 1980. At the first conference industry was extremely critical of Computer Science Departments for producing graduates with a number of unconnected skills which had little or no relevance to the needs of commerce. At the second conference a greater appreciation of the type of training given to the Computer Science graduate was achieved however the relevance of this training was still hotly debated. One of the points raised by Prof G.Murray at the Rosebank Conference was that the needs of industry were not well defined whereas the international requirements of a Computer Science course was. This meant Computer Science departments tended to play safe and produce internationally recognised graduates rather than locally useful ones.

Since 1980 the Computer Users Council has established itself as a force in the South African computer scene. It has accepted the responsibility to represent the needs of its members - many large D.P. organisations. It regularly polls its members on their requirements and, through meetings with university representatives conveys the needs of the industry. No longer can it be said that the needs of industry are not well defined. At a recent meeting at UCT with the local branch of the CUC I was emphatically informed that the industry considered it would have a continuing need for Cobol programmers in the foreseeable future and did not see this being significantly eroded by fourth generation languages. Whether one agrees with this or not at least commerce is speaking with one voice.

Since the conference of 1973 changes have also occurred in the academic computing community. Some universities have established separate departments of Business Data Processing. These departments do not profess in any way to teach Computer Science but are free to teach information systems. The only constraints on these departments comes, perhaps, from the accounting departments, which sired, some of these departments and seems to have kept a paternal say in ensuring that the needs of the accountants remain well catered for. In addition the Technicons have introduced Computer Science courses. These courses are designed as training courses catering to the needs of commerce. These courses should not be teaching Computer Science, in my opinion, but be wholly committed to the IS syllabus and producing commercial analyst programmers.

With the Business Data Processing or Information Systems Departments and the Technicons producing commercially orientated analysts and programmers the skilled needs of commerce should be adequately covered.

The question remains, with no computer manufacturing in South Africa, and a relatively small systems software industry, for whom are the Computer Science Departments producing their graduates, and indeed is there a need for Computer Science in South Africa?

THE FUTURE OF COMPUTER SCIENCE IN SOUTH AFRICA

Let us first consider who we regard as a computer scientist. It is a commonly accepted view that a three year degree does not produce a person trained in any specific discipline but having been introduced to some topics at intermediate level. The specialist is only produced as a result of a four year degree or B.Sc.(Hons) or higher degree. A fully qualified Computer Scientist is thus a graduate with four years instruction in Computer Science.

For whom are we producing these highly qualified graduates? Despite its lack of size South Africa does have a viable software industry producing real-time systems, micro computer software, communications software, supporting and developing databases etc. In these areas the advanced techniques of software design, concurrency and algorithm analysis are important. Many commercial companies state that they are not concerned about the execution speed of programs and are prepared, if necessary, to purchase additional computer power rather than spend time optimising programs. However, in general, a company will purchase a new computer to execute a poorly designed applications system but will not accept an inefficient operating system or a slow teleprocessing monitor. If the needs of this software industry is not met by Computer Science Departments than by whom will it be met? Not by information systems graduates. Certainly not by commercial programming colleges. Rather this void will be filled by other university departments whose sole concern is to climb on the computing bandwagon and increase their student numbers. In the van of this group are the electrical engineering departments who are in some cases exploiting the phrase "software engineering" to motivate that software should be designed by engineers. This group is already making claims that they are doing as much Computer Science as the Computer Science Departments. However good the intentions of these engineering departments, and however good their software courses they are restricted by the requirements to produce professional engineers that they are unable to include adequate ancillary courses in their already crowded syllabus to give the required environment necessary for a Computer Scientist.

Do we require to produce as many graduates as at present? There are twenty one universities in South Africa, of which twenty have Computer Science Departments. At one time I felt this was an excessive number for a country of our size, however, this year while attending a conference in America I was amazed to hear that that there are 25 degree granting institutions in the Boston area alone. The number of qualified graduates produced in South Africa is relatively few, less than 100 per annum. Of this elite group approximately twenty percent do not choose to enter local industry but elect either to remain in academic environments, in South Africa or overseas, or choose to leave the country. Thus only some 80 fully qualified graduates are entering south African industry each year.

THE FUTURE OF COMPUTING

The computer industry world-wide is facing an era of tremendous change. The converging technologies of computing and communications coupled to the innovations in the manufacture of components has heralded the introduction of the era of information technology. The changes which are expected to appear over the next decade will change computers from pure data processing machines to knowledge processing machines. The basic concept of the fifth generation Computer Systems is described as [10] 'The Fifth Generation Computer Systems will be knowledge information processing systems having problem-solving functions of a very high level. In these systems, intelligence will be greatly improved

to approach that of a human being, and when compared with conventional systems, man-machine interface will become closer to the human system".

This research project in which the Japanese government is investing US$500M and its counter part projects in America and Europe all have the common aim of bringing the computer more in contact with the average man. This requirement for ease of use will undoubtedly introduce larger and more complex software products. While it is not possible for South Africa to compete with the research aims of the developed countries, Europe is planning on 9268 man years of effort between 1983-1993 [11], it is necessary for some qualified resource to be capable of understanding the functioning of the advanced systems to support them locally and where possible adapt them to the local South African requirements. These people are Computer Scientists. Without this local expertise South Africa will be totally dependent on imported knowledge in one of the most strategic areas.

THE COMMERCIALISM OF COMPUTER SCIENCE

Recently a trend has appeared in Computer Science Departments in South Africa of producing information systems graduates in Computer Science departments under the guise of Computer Scientists. This, I believe, is an unfortunate step. The information systems curriculum stresses that the disciplines belong to different environments. It is difficult to appreciate how a science faculty will permit the necessary number of non-science courses such as personnel management, accountancy business methods, economics, which should be included in an information systems course, to be included in a science degree. This can only lead to a graduate being produced who is neither a computer scientist nor an information systems graduate. He will be partially trained in both disciplines. This practice can only lead to greater confusion in the market place about the usefulness of computer science graduates than already exists at present. These departments should rather change their name to information systems departments and move to the commerce faculties of their respective universities where they can produce a more effective information systems graduate.

CURRICULUM CONTENTS

The contents of the computer science curriculum should be based on pure theoretical foundations. It should contain program verification, top down design, programming environments, programming languages, compilers, operating systems, data structures, data bases and data models, mathematical theory of computation, analysis of algorithms, computer systems modelling, concurrent languages, architecture, and artificial intelligence. All of these topics should be treated from a reasonably formal approach rather than purely explaining the uses of the techniques. The South African Institute of Computer Scientists is currently considering the accreditation of computer science degree courses. I sincerely hope it will distinguish between information systems and computer science in its assessment and provide to commerce a guide to the particular strengths of the products of the various universities.

CONCLUSION

In most countries computing and particularly computer science is regarded as being of strategic importance. South Africa can support and requires strong Computer Science community producing a relatively small number of highly qualified graduates skilled in the techniques of producing well designed, efficient and well documented computer systems, using the most advanced software techniques. The challenge of

producing this pool of qualified manpower rests entirely with the computer science departments. It must rise to this challenge of producing a universally acceptable product and not be sidetracked by short term attempts at commercial acceptability.

BIBLIOGRAPHY

1. Curriculum Committee on Computer Science, Curriculum '68, Recommendations for academic programs in computer science. *Comm ACM* 11,3 (March 1968), 151 - 197.

2. Curriculum Committee on Computer Science, Curriculum '78, Recommendations for academic programs in computer science. *Comm. ACM* 22,3 (March 1979), 147 - 165.

3. Traub J.F.(Ed), Quo vadimus: computer science in a decade. *Comm. ACM* 24,6 (June 1981), 351 - 367.

4. Ashenhurst R.(Ed), Curriculum recommendations for graduate programs in information systems. *Comm. ACM* 15,5 (may 1972), 364 - 398.

5. Cougar J.D.(Ed), Curriculum recommendations for undergraduate programs in information systems. *Comm. ACM* 16,12 (December 1973), 727 - 749.

6. Nanumaker J.F. et al(Eds), Information systems curriculum recommendations for the 80s:Undergraduate and graduate programs. *Comm. ACM* 25,11 (November 1982), 781 - 805.

7. IEEE Computer Society, Curriculum recommendations for software engineering. 1982.

8. Brittan J.N.G.(Ed), An International curriculum for information system designers. IBIICC 1974.

9. DPMA, DPMA's model curriculum. Oct 1980.

10. Moto-oka t. et al, Challenge for knowledge information processing systems in *Fifth Generation Computer Systems*. North Holland Publishing Company 1982.

11. -, Proposal for a council decision adopting the first European programme for research and development in information technologies. *Com(83)* 258 June 1983.

High-level Debugging Systems for Pascal: Interpreter versus Compiler

MICHAEL HIRSCH
Department of Applied Mathematics,
The Weizemann Institute of Science,
Rehovot,
76100 Israel

STEPHEN R. SCHACH
Department of Computer Science, Vanderbilt University,
Nashville, Tennessee 97235, U.S.A.

WILLEM R. VAN BILJON
Computer Science Division, NRIMS, CSIR
Pretoria, 7700 South Africa

ABSTRACT

Two approaches to high-level debugging systems for Pascal are compared, namely the use of a debugging precompiler as against a debugging interpreter. A description is given of a high-level Pascal debugging interpreter which has the power of the precompiler approach, but with none of the disadvantages.

INTRODUCTION

A *high-level* debugger is one whose output reflects the high-level language in which the program being debugged was written. Three systems that support high-level debugging of Pascal programs are the PASCAL-I program development environment[1], HEAPTRACE, a portable trace for the Pascal heap[2], and the Pascal compiler of Watt and Findlay[3] which provides forward and retrospective trace facilities and a post-mortem dump. A *very high-level* debugger displays a user's data structure in a format as close as possible to the way he conceptualizes that data structure, be it a tree, or a doubly linked list; GRAPHTRACE, a very high-level trace for the Pascal heap[4], is an example of a very high-level debugger for Pascal.

The systems of Cichelli, and of Watt and Findlay, have a drawback in that neither supports dynamic variables. PASCAL-I is an interactive program development system for Pascal-S[5], a subset of Pascal that excludes dynamic (heap) variables, while Watt and Findlay's high-level post mortem dump includes only static variables. A further disadvantage of PASCAL-I is that in order to to make use of of the wide variety of powerful features provided, the user has to learn a fairly extensive command language.

While GRAPHTRACE and HEAPTRACE specifically provide high-level debugging of the heap, they suffer from major limitations as a consequence of the fact they are written in standard Pascal (in order to achieve true portability) and as a result of the implementation strategies chosen. In both packages the user inserts debugging commands (in the form of comments) within his source code, which is passed through a precompiler. The Pascal output from the precompiler is then compiled and linked in the usual way. Tracing of the heap is achieved as follows: whenever the user creates (via the **new** command) a record to be traced, the system creates a "hyper-record" which is two-way linked to the

corresponding record. The hyper-records themselves are two-way linked. In this way the system can access any user defined record, even if the user has mismanaged his pointers.

There are four major problems with HEAPTRACE and GRAPHTRACE.

1. No information can be gained from either system after a run-time failure as, in order to achieve portability, both systems are dependent on the Pascal run-time environment.
2. No interactive modification of the trace commands is possible. The commands are inserted into the source code at places where the user surmises that diagnostic information would prove helpful; if the user now wishes to obtain information from elsewhere in the program, he must go through the edit-precompile-compile-link cycle.
3. Both systems disable the (non-standard) garbage disposal procedure **release**, because the portability constraint requires the hyper-records to be stored on the heap itself, and hence no interference with the structure of the heap can be permitted.
4. Finally, the problems of uninitialized variables in general (and pointers in particular) require the user to initialize all fields of records to be traced.

In this note we describe a debugging system that solves problems 1) to 4) above, while still providing high-level debugging of all variables, including heap variables. In addition, the user need not learn a complex command language; debugging commands are almost all specified in the form of Pascal statements. Finally, the system is fully portable, being written in standard Pascal.

SYSTEM OVERVIEW

The debugging interpreter supports all the high-level heap debugging features of HEAPTRACE[2], as well as conventional high-level debugging tools. These include a line-by-line execution trace in which each source line is printed before it is executed, and a control flow trace (at each branch in the program the source line is printed, and information as to which branch was taken is given). There is also a variables trace; for each l-value, the appropriate source line, the scope of the variable, its name, old value and new value are printed.

The system consists of the portable P compiler[6], minimally modified, together with a debugging P-code interpreter. The compiler generates P-code from the user's Pascal source code. The source code, P-code, and compiler symbol table are all passed to the interpreter. Hyper-records are maintained in a similar fashion to HEAPTRACE and GRAPHTRACE, but on a separate heap. The user interacts with the system at run-time to enter debugging statements expressed in Pascal.

This method solves all four of the above problems. Firstly since the interpreter flags all run-time errors when they occur, the user may enter any debugging statements after his program has crashed, including the equivalent of requesting a selective post-mortem dump. The debugging statements can be entered either in the source code, or interactively at run-time, or both, thus solving problem two. The separate heap for the hyper-records solves the third problem; if a record is deleted from the program heap via a **release** command, the heap of hyper-records is appropriately modified. Finally, the interpreter is able to distinguish uninitialized scalar variables thus enabling the user to trace or to dump uninitialized variables.

The system has no explicit debugging language. It is assumed that a programmer debugging a Pascal program is already familiar with Pascal, and thus it is the obvious command language. Typical commands are:

```
writeln(employeeptr↑.socialsecuritynumber);
```

or (to change the values of two variables)

```
begin
        y2:=1;
        y3:=2

end;
```

However, it was found that Pascal is inadequate for certain debugging functions, especially examining the heap. The decision was therefore taken to augment Pascal by ten additional standard procedures, such as **breakpoint** (to effect a suspension of execution at a desired point), and **heapdump**.

IMPLEMENTATION NOTES

The system effectively consists of three components, the compiler, the interpreter, and the debugger. The compiler generates P-code (for a hypothetical stack computer) from the user's source program when called by the interpreter, and from the debugging commands when called by the debugger. The interpreter simulates the hypothetical stack computer. The debugger is invoked by the interpreter whenever a run-time exception occurs.

The P compiler was extended to accept the ten additional standard procedures, and to generate two additional stack computer instructions, namely LNO (line number), generated at the start of each statement and after each condition, but before the conditional jump, and CDP (call debugging procedure), generated by each of the additional standard procedures. CDP is analogous to CSP (call standard procedure). Finally changes were made so that the system runs in compile-and-go fashion.

The interpreter is based on the hypothetical stack computer model. Extensions were made to this model to allow for a simple tagged architecture, to monitor the heap, and to interpret the added instructions.

The architecture of the stack computer has three main stores, namely the data store, the code store, and the hyper-heap (where the hyper-records are stored), as well as six registers. The registers are Stack Pointer (SP), Program Counter (PC), Markstack Pointer (MP), Extremestack Pointer, New Pointer and Line Number. The SP, PC and MP registers are implemented as value parameters of the interpreter, so that recursive calls on the interpreter implicitly permit the enviroment to be saved.

The data store is implemented as an array of variant records, with the type of that word of store (integer, character, etc.) as tag field. Two Boolean flags are provided, one to denote if the variable is undefined, the other to indicate whether the word in question is being traced. The data store contains the stack growing up from 0, and the heap growing down. The code store is maintained separately from the data store.

The hyper-heap is implemented as an array of hyper-records. The fields of a hyper-record include a pointer to the start of the corresponding structure on the heap (i.e. in the data store), and a pointer to the record in the symbol table containing the type information for that structure.

The task of the debugger is exception handling. It reports the exception to the user, then reconstructs the display from the static link,

and calls the compiler to solicit a Pascal statement from the user. Finally, it calls the interpreter recursively to execute this statement. If possible, execution of the user's program is then resumed.

RESULTS AND CONCLUSIONS

The system has demonstrated that it is possible to avoid the major drawbacks of Pascal high-level debugging systems based on precompilers by using an interpreter. The system is portable, and uses Pascal as command language, thereby obviating the need for the user to learn a separate command language.

It must be pointed out that, despite this success, the debugger-interpreter has some disadvantages of its own. As a consequence of the decision to make only minimal changes to the P compiler, certain limitations have been forced on the system. Firstly since the P compiler treats enumeration types as integer constants, no type information is on hand when certain errors occur. Insufficient information is available to supply high-level debugging information for handling a 'value out of range' error (generated by instruction CHK) or a 'no case provided for value' error (generated by UJC); the relevant enumeration value cannot be displayed without the introduction of a third additional P-code.

More serious is the fact that the compiler does not support any tagged features, forcing items to be traced or debugged by address. When tracing an array, the address of the array compiles to the same address as its first element, causing only the first element to be traced. Similarly, when dumping structures, the largest possible structure starting at the given address is printed. A request for a dump of the first element of an array will thus result in the printing of the whole array. Analogous action is taken in the case of records.

However, these problems can be solved by writing a Pascal interpreter (or an interpreter for an abstract syntax tree representation of a Pascal program), rather than modifying an existing compiler. All data type and lexical information can then immediately be made available.

On the grounds of our experiences with precompiler-based Pascal debuggers as well as the debugging interpreter described above, it appears that a Pascal debugging interpreter can provide all the facilities of a precompiler-based debugger, but with none of the disadvantages. It remains to be seen whether this result is equally applicable to other programming languages.

ACKNOWLEDGEMENT

The work of Michael Hirsch and Willem van Biljon was supported (in part) by the South African Council for Scientific and Industrial Research, Pretoria.

REFERENCES

1. R. Cichelli, Pascal-I - Interactive, conversational Pascal-S, *Assoc. Comput. Mach. SIGPLAN Notices*, 15(1), 34-44 (1980).

2. S. R. Schach, A portable trace for the Pascal heap, *Software - Practice and Experience*, 10, 421-426(1980).

3. D. A. Watt and W. Findlay, A Pascal diagnostic system, in *Pascal: The Language and its Implementation* (Ed. D. H. Barron), Wiley, Chichester, 1981.

4. S. L. Getz, G. Kalligiannis and S. R. Schach, A very high-level interactive graphical trace for the Pascal heap, *IEEE Trans. Software Engineering*, **SE-9**, 179-185 (1983).

5. N. Wirth, Pascal-S: A subset and its implementation, in *Pascal: The Language and its Implementation* (Ed. D. H. Barron), Wiley, Chichester, 1981.

6. K. V. Nori, U. Ammann, K. Jensen, H. H. Nageli and Ch. Jacobi, Pascal-P implementation notes in *Pascal: The Language and its Implementation* (Ed. D. H. Barron), Wiley, Chichester, 1981.

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 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 submitted in triplicate to:
Prof. G. Wiechers at:
INFOPLAN
Private Bag 3002
Monument Park 0105

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 an abstract 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 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 I, 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. BOHM,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 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 to 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.

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.