QI QUAESTIONES INFORMATICAE

Volume 4 Number 1 January 1986

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

Subscriptions

Annual subscriptions are as follows:

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

Circulation and Production

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 Science Department of the University of the Witwatersrand and printed by Printed Matter, for the Computer Society of South Africa and the South African Institute of Computer Scientists.

Ways of Assessing Programming Skills

J.M.BISHOP
Computer Science Department
University of the Witwaterand, Johannesburg
2001

ABSTRACT

The programming skills taught in first year computer science courses are more difficult to assess than the skills in older science courses because of the very large numbers of students involved coupled with material which is non-quantitative. For any question, there is a variety of solutions, and these are of a prose rather than a numerical nature. Moreover, writing a program involves a certain amount of design, the time for which is not easily fitted into traditional examination methods. Nevertheless, effective questioning and accurate marking is vital to the success of such courses. This paper looks at traditional and novel ways of assessing programming. It relates these to an accepted taxonomy of educational objectives and gives several practical guidelines for improving the quality of assessment.

INTRODUCTION

First year computer science courses exhibit a combination of properties which make them very difficult to assess. These are :

- large numbers of students
- practical and theoretical components
- the "prosiness" of programs
- the variety of solutions possible for any one question
- the "design factor" inherent in programming.

Other science courses have some but not all of these properties, and have built up ways of coping with them over several decades. Computer science courses have been going for only some fifteen years and as yet no body of experience or even opinion exists as to how best to assess them. It is possible that computer science at this level has more in common with the humanities, and can draw from their experience. Certainly, except for the practical component, English I is faced with much the same problems.

Computer science at first year level does not consist only of programming. Usually there are topics on machine organisation, data structures or system software as well. However, these lend themselves more to traditional examination techniques. This paper, therefore, concentrates on ways of improving the assessment of programming skills, although the guidelines on how to examine apply generally.

THEORETICAL vs. PRACTICAL

Programming is a practical activity, performed with equipment and occupying real time in doing so. In order to learn to program one must practise it, and certainly any computer science course would require students to complete a number of practical assignments. Whether or not those assignments are then marked and used as part of the final assessment is open to debate. Those in favour of including them (among which are most of the students) argue that assignments represent a realistic programming effort while the necessarily small programs in an examination are not an accurate assessment of a student's ability to program. Those against including them claim that cheating is rife and that it is too difficult to assign accurate marks to assignments. They would rather see assignments used as a modifier in borderline cases.

Whatever the pedagogical justification may be, it seems as if the trend in academic staff-student relations is towards continuous assessment, in other words assignments must count. Given this, the academic's aim must be to eliminate the negative factors - cheating and inaccurate marking - as far as possible. Several practical ways of doing this are discussed in Part B below.

The proportion of the final mark assigned to this continuous assessment will vary from course to course, but most faculties insist on at least 50% of the mark coming from a formal examination written at the end of the course. It is therefore most important that the written

examination is fair and effective and that the marking is accurate and consistent. By looking outside computer science to education and by taking into account the practical experience of many universities, I have been able to identify factors which affect the quality of formal examining. These can be grouped into those relating to the *effectiveness of questioning* i.e. does the examination reflect the course material and test the skills we want the students to have gained; and those which aid the *accuracy of marking*. Essentially, this latter group involves practical tips on the format and style of papers. It is interesting that some of these will also make the examination seem easier for the student to write.

The rest of the paper is therefore divided into two parts, considering the theoretical and practical aspects separately. However, there are inevitably points which apply to both sides, and the careful reader should be able to appreciate these.

PART A - THEORETICAL ASSESSMENT

EFFECTIVENESS OF QUESTIONING

To be effective, an examination must reflect the objectives of the course. Unfortunately, those objectives are seldom stated or if they are, they are not couched in terms of educational skills, but rather as specific achievements e.g. "at the end of this course, a student should be able to write, debug and document a program of up to 500 lines". Educational skills are identifiable, and an excellent taxonomy is given by Bloom [1]. He lists six skills in order of conceptual difficulty i.e.

knowledge
comprehension
application
analysis
synthesis
evaluation

Each question in an examination can be placed into one of these categories and the percentage of the whole examination devoted to each skill determined. The examiner can then ask himself whether this is an accurate reflection of the course's perceived objectives and perhaps adjust the questions accordingly.

Looking at the questions which are typically asked in first year programming examinations we find they fall in seven groups.

1. Facts about the language.
2. Spot the error.
3. Evaluate expressions.
4. Produce the output from a program
5. Give an isolated declaration or construct.
6. Write a program or subprogram.
7. Compare the use of constructs.

If we relate these to Bloom's taxonomy, we find that they span the whole range of educational skills. Matching them to the taxonomy, we have :

SKILL	**QUESTION GROUPS**
knowledge	1
comprehension	1, 2
application	3, 4
analysis	4
synthesis	5, 6
evaluation	7

Investigation on the frequency of the questions in each group, reveal that our examining is heavily biased to the upper skills, specifically application, analysis and synthesis. Is this really what we want to examine? In a first course, should we not give some credit for the acquiring of lower level skills? No matter how simple the programs, a paper consisting solely of "Write a program" type questions seems pitched at an overly high level - unless the programs are "seen", having been done in tutorials, in which case these questions reduce to the simplest skill, knowledge.

EXAMPLES OF LOWER SKILL QUESTIONS

To promote the use of questions in the lower skills, here is a selection of good ones taken from actual examination papers.

1. Facts.

Q. State two advantages of using named constants.

Q. Explain what is wrong with the boolean expression
(number * 10) > maxint.

There are many similar questions which can be put at the start of a paper and serve to get the student going and give him confidence.

2. Spot the error

Q. The following program is not intended to be meaningful, but it is also not correct Pascal. Correct all of the mistakes and give your reason(s) for each correction.

```
PROGRAM index(input, output)
   VAR i,j : INTEGER;
        r,s : REAL;
   BEGIN
      j := 3.4;
      read(i, k, r),
      r + j := r;
      If r>7.3 OR 5<i then s := 4.7;
      writeln(i, j, r, s);
   END.
```

This kind of question is not very popular, since many would claim that spotting compilation errors is an incorrect means to an end. The end is the proper use of types and operators, usually, and this can be examined by questions in the next group.

At Wits, we had spot the logical error questions for two years but found that the students fared badly. The trouble is that you either spot the mistake immediately or not at all, and this is unfair examining practice. Readers can test themselves on the following sample:

```
{Read and write 10 numbers}
FOR n:=1 to 10 do
   read (n);
   write (n);
```

3. Evaluate expressions.

Q. What is the value of

```
substr ('ECOLOGY', index('ECOLOGY', 'LOG'));
```

Q. Give the value of I in each case:

```
I := 297/50
I := MOD (33,8)
I := INT (3.59 + 2.74)
```

Q. State the type of the value produced by

```
3 + 4/2
```

These questions are an excellent way of testing knowledge of types and operators. They can be carefully designed to pick out important cases without degenerating to the level of trick questions.

4. Produce the output.

Q. What is the output from the following program?

```
PROGRAM passby (output);
   VAR i, j, k : integer;
      PROCEDURE assign (i :integer; var j : integer);
        VAR k : integer;
        BEGIN
           i := 1;  k := j;  j :=2;
           writeln (i, j, k);
        END;
```

```
BEGIN {passby}
   i := 3;  j := 4;  k := 5;
   assign (j.i);
   writeln (i, j, k);
END.
```

This kind of question is usually used to examine parameter passing or scope rules. The above is an example of the skill of application; it can have an element of analysis if the student is also asked to explain why the answer was arrived at.

FAIRNESS OF QUESTIONS

An examination must reflect the objectives of the course but the way in which the questions are asked must also relate to the class's experience. In other words, one cannot teach the first five skills and then spring an evaluation type question on the students in the examination. More subtly, one should not use any of the kinds of questions given above unless the students have practised these activities in class or tutorials or even in tests.

For example, it is unfair to set a "produce the output" type of question unless students have specifically seen "bench-checking" done in class. They would otherwise not know to set up little boxes to aid them in following the execution of the program. Just to assume that "they must have done it or how could they have debugged their programs" is not good enough.

A most important point is that one should not consciously hide information from students by using ridiculous identifiers or omitting comments. It is better to have program excerpts which are realistic, with their proper meaningful identifiers. This is closer to the course objectives and will enable students to spot errors or dry-run programs much more easily.

ACCURACY OF MARKING

Looking back at the seven question groups, we see that the first five can usually be phrased to have very short, specific answers - even a single word or number. The crunch comes at 6 - write a program. To achieve accurate and consistent marking of the programs produced in examinations, one can employ several variations of "divide and conquer".

1. Step-by-step.

Here the problem is broken down into parts and the student is led through a top-down, stepwise refinement exercise. For example :

```
Q. A coin collector wants at least one of each of the
   denominations of British coins (i.e. 1p, 2p, 5p, 10p, 20p,
   50p, £1.00) for his collection.
   (a)   Define a symbolic type for each denomination.    (4)
   (b)   Define a set type which encompasses all the
         values of the above symbolic type.               (2)
   (c)   Write a procedure which will write out a
         given value of the type defined in (a).          (7)
   (d)   Write a procedure to write out all the
         values contained in a set of the type
         given in (b), using the procedure in (c).        (7)
                                                    [20 marks]
```

The advantages of breaking down the problem are that the student is led to the desired solution - thus minimising the possible variations - and the design factor is very largely removed. Compare the answers one would expect to the above myriad possibilities for this question

```
Q. Write a Pascal program to read a date and determine the
     date exactly nine months thereafter.
```

The disadvantage of doing much of the design for the student is that he is forced into a structure with which he may not be comfortable. It may not have been the way he would have solved the problem and so he does not easily perceive what he must do. A side-effect of this method is wordiness - the questions are long and the student has much reading to do. The bad effects of this can be minimised if a conscious effort is made to be brief, and if the sentence containing the actual question is underlined so that it stands out.

2. Providing initial environments.

It is becoming more and more popular to specify part of a solution, and get the student to complete it. By giving all or some of the declarations, the question can focus on the action. For example :

Q. A common problem in data processing is the merging of two serial files, each already sorted in ascending order, to form a third serial file, also in ascending order of the data items. Show how this could be done in standard Pascal, by completing the program below.

```
PROGRAM mergefiles; {merge one and two to produce three}
TYPE datafiles = file of integer;
   VAR one, two,  {the original files}
       three      {the merged file} : datafiles;
   BEGIN
      reset (one);
      reset (two);
.....
```

The advantage of this technique is that the marker has a standard terminology in the solution. More experience with this method is reported in [2].

3. Exempt I/O and documentation

Some examiners feel that the input/output required for a proper interactive program is too much to expect in an examination, and does not the student's knowledge of programming. The same goes for documentation. One sees at the front of a paper "Programs written under examination conditions are not expected to be user-friendly or commented".

The obvious advantage to the student is that it saves time. However, a subtle disadvantage is that he may be unused to writing programs any other way and so is thrown off balance (would this were so!). Perhaps a better way of getting rid of I/O is to limit the questions to the writing of subprograms.

4. Overall format.

One of the simplest ways of greatly reducing the marking time for examinations is to have the students write on the paper itself. In this way, all the questions are in order and the students are encouraged, by the space given, not to write too much. This is very effective when coupled with the other methods discussed above.

5. Multiple choice.

A time-honoured method of standardising marking is the use of multiple choice questions. These would be applicable in the first 5 groups of questions, but cannot cater for the synthesis group. However, unless one is going to mark these questions by computer using mark sense cards, there is really nothing to be gained by giving multiple answers over simply having a space for the answer to be written. An interesting assessment of the use of multiple choice for computer science is given in [3].

PART B - PRACTICAL ASSESSMENT

If practical assignments are going to be counted towards the final mark, then both students and staff must feel confident that

- cheating is minimised
- marking is accurate and consistent

The first goal is achieved by carefully organising the setting and marking in advance. The second goal can be brought closer by drawing from the experience of the humanities. These are discussed in turn.

MINIMISING CHEATING

In classes of over 300 students, catching a case of cheating seems impossible. There is also doubt as to what constitutes cheating [4]. The one kind of cheating we cannot catch is that of a

student getting a senior to do his work. The other two kinds - copying from a book and copying from another student in the class - can be detected by use of the following methods.

1. Set several versions.

For any one assignment, set several (say 3 to 6) different versions, and allocate these at random to the students. The chances of co-workers getting the same assignment to do are alim. Then all solutions to one version are given to a single marker. He will be able to mark consistently, and to be able to detect similar solutions, which are then examined for copying.

2. Mark quickly.

Ask each marker to mark his exercises within a few days. This increases his chances of recognising patterns he has seen previously, and contributes towards more consistent marking.

3. Create new exercises.

Don't use stereotyped exercises which are likely to be solved in books. Be creative, even it takes longer.

At Wits, we have followed this procedure for three years, and each year have caught one or two cases of copying in the early assignments. The wide, though anonymous, publicity given to the incidents tended to discourage cheating attempts from then on.

The difficult bit is ensuring that the versions all test the same things, and are of the same standard. As an example of how this can be done, Appendix I gives the three questions for our second 1984 assignment. These exercises were started after five weeks of Pascal lectures and handed in three weeks later.

ACCURATE MARKING

The problem of consistent, fair marking of assignments is very similar to that of marking English essays and is well set out in one of the few papers in the literature [5]. The authors adapted a grading method from the humanities called the Diedrich Scale. A set of criteria is agreed upon by all markers (or decreed by the lecturer) for a particular assignment. Then a scale of marks is listed alongside each criteria, with different ranges reflecting the differing importance of each. For an essay, the criteria Diedrich established are :

Ideas	2	4	6	8	10
Organisation	2	4	6	8	10
Flavour	1	2	3	4	5
Wording	1	2	3	4	5
Usage	1	2	3	4	5
Spelling	1	2	3	4	5
Manuscript Quality	1	2	3	4	5

For each script, the marker circles a mark on each row and does not total the mark, lest he be influenced by a run of good marks to tighten up or vice versa.

Hamm *et al* feel that the criteria should vary from assignment to assignment and describe a program that will produce scales such as the above, for given criteria. For example, in initial assignments, layout of the program may count, but later on it will become irrelevant and other factors such as the use of procedures or the report become important.

A similar method has been used at the University of Southampton and Wits since 1975, except that it has the added advantage of using words to describe the values for the criteria rather than numbers. For example, if testing is a criterion then we might have

```
testing = (all cases tested, some important ones missing, inadequate)
```

The marker is led to look for the precise qualities - or lack of them - that feature in the objectives of the exercise and of good programming in general. Once the words have been chosen, a separate pass is made where values are assigned to each word and totals reached. In a final moderating phase, the values may be adjusted slightly to get a better distinction between good and bad solution.

The marking scheme used for the assignment in Appendix I is given as Appendix II.

IS IT WORTH IT?

There is a world of difference in the effort required to set an assignment which consists of an exercise out of the text book and is marked by gut reaction or a percentage (e.g. "worth a first - 75%"), and the procedures described above, but if we are serious about achieving a reliable method of assessment, the above procedure seems to work well. Its evident advantages are :

- minimising cheating
- more consistent marking across markers
- objective-oriented marking
- use of the full percentage scale
- student confidence in marking.

The last two deserve more discussion. If the traditional system of assigning a single mark for the whole program is employed, it is unlikely that this will go higher than 8/10 or 18/20 or 80%. It is human nature to regard all work as imperfect and to shy away from assigning a perfect mark to it. Using a part scoring system with definite marks for definite things enables a student to get marks where due, and it is common for many to get over 80%. For the assignment shown in the Appendix, of the 170 students 57, i.e. one third achieved over 80%. By the same token, some marks will always be achieved by even appalling work.

As far as the benefits to the student go, there is a difference of opinion as to whether students should see the marking schemes or not. At Jacksonville [5], they do, and as a result the students gear their efferts towards obtaining good marks in each category. Staff are also faced with questions such as "What must I do to get full marks for testing?", the answering of which is no doubt good for their souls! At Wits, we have shied away from the total disclosure approach, in the belief that students should have a fresh and constructive attitude to their programs, rather than a mark oriented one.

A particular consideration is also that sometimes the scheme is only set up once the assignments are in and the lecturer can see how the students coped. (Although solutions can be prepared in advance for small assignments, it is unusual to do so for the larger ones of 500+ lines). So, students are told informally what kind of thing we look for in a good program, and they may be given the breakdown of their mark on request.

CONCLUSION

This paper has ranged over a wide area of assessment and has posed the problems and suggested solutions. The overall conclusion one can draw is that effective, fair and consistent assessment can be achieved but it requires some effort. I hope the experience reported here and the guidelines given will encourage others to exert such an effort.

REFERENCES

[1] Bloom B.S., *Taxonomy of educational objectives : Book I Cognitive Domain*, Longman, London, 1956.

[2] Trombetter M., On testing programming ability, *SIGCSE* ***II*** (4) 56-60, December 1979.

[3] Van Meer G. and Dodrill W.H., A comparison of examination techniques for introductory computer programming courses, *SIGCSE* ***15*** (4) 34-38, December 1983.

[4] Shaw M. *et al*, Cheating policy in a computer science department, *SIGCSE* ***12*** (2) 72-76, July 1980.

[5] Hamm R.W. *et al*, A tool for program grading: the Jacksonville University Scale, *SIGCSE* ***15*** (1) 248-252, February 1983.

ASSIGNMENT IN PASCAL

General Instructions

1. Each assignment concerns inputting text and outputting it in some modified form. Your program must be able to read several paragraphs from the keyboard or a file.
2. Paragraphs are ended by a blank line. Do not add any other data terminators to the text - make use eoln and eof.
3. Do not use arrays - they are unnecessary.
4. The problems can be solved in about 60 lines.

1. Comments.

Write a program which reads in paragraphs and prints them out again, ignoring all text between Pascal comment brackets {}. Print suitable warning messages if

- a { is found inside a comment
- a } is found without a {
- the paragraph ends without a matching }.

At each end of the paragraph, print the percentage of text (excluding blanks) that occurred in comments.

Example:

Input	Output
This is the same length as the comment {This assured comment is the same length as the rest} rest assured.	This is the same length as the comment rest assured. Comment is 50% of the text.

2. Standards

Write a program which reads in paragraphs and prints them out again, having made any changes required to have the text conform to the following standards:

- Five spaces at the start of a paragraph
- Capital letter at the start of each sentence
- Two spaces between sentences or main clauses

Example:

Input	Output
It is always hard to type to a standard.Some people use different ones: single or double or no spaces before a sentence. i think double looks best.	It is always hard to type to a standard. Some people use different ones: single or double or no spaces before a sentence. I think double looks best.

3. Pig Latin

A traditional children's code consists of taking the first letter of each word, putting it at the end of the word and adding "ay". Write a program to read in paragraphs and print them out again in Pig Latin, with the same spacing and lines. Take care of the following exceptions:

- vowels at the start of a word don't move
- a capital at the start of a word becomes small and the second letter becomes a capital.

Example:

Input	Output
Latin is a language that no gentleman should know, but that every gentleman should have forgotten.	Atinlay isay aay anguagelay hattay onay entlemangay houlsay nowkay, utbay hattay everyay entlemangay houldsay avehay orgottenfay.

PASCAL ASSIGNMENT MARKING SCHEME

PROCEDURE

1. Read the report and assess it.
2. Look at the output and assess the first two categories on it alone. Pay particular attention to whether they solved exatly what was asked - no marks for more, marks off for less.
3. Assess the next categories on the program.
4. Enter the category scores on a list, add up and multiply by 5 for percentage.

NOTES

1. Please put comments on the programs, indicating where marks were lost.
2. Check the testing carefully - they must catch every case.

SCHEME

Results	7	All correct and program does all required
	4-5	Misses some important cases
	3	Can't handle basic eoln or paragraph end
	2	Incorrect answers or ring buffer overflow
	0	None
Testing	3	All cases tested
	1	Some important ones missing
	0	Inadequate
Algorithm	4	Competent i.e. few loops (2 is enough) no repetition, structure clear
	2	Repetition and hard to prove correct
	1	Really messy
Control structures	2	All correctly used
	1	Messy use of sets, no else's, too many conditions on loops (i.e. no functions) etc.
	0	All of these
Procedures & Scope	3	Correctly and effectively used
	2	Too many parameters or none where needed or too many variables.
	1	Not used - these problems cry out for them.
Report	1	Fine
	0	Poor

Book Review

Muller, R. L. and Pottmeyer, J. J. (editors). *The Fifth Generation Challenge* Proceedings of the ACM Annual Conference (ACM '84), North–Holland, Amsterdam, 1984. 335 pages. ISBN 0 444 87661 8.

review by Philip Machanick, Computer Science Department, University of the Witwatersrand, Johannesburg.

Despite the title, the conference did not only cover the Fifth Generation. There were many worthwhile papers about artificial intelligence, computing in the home, software engineering, system design and education.

Fifth Generation sections include *Projects Around the World*, *Principles of Knowledge Based Systems*, *Case Histories*, *CAD/CAM and Machine Perception*, *Special Architectures for Fifth Generation Systems* and *The Impact and Issues of the Fifth Generation.*

Unfortunately, the Proceedings were produced for the conference and include (now) irrelevant details like the conference time–table. For the same reason, many potentially interesting papers were published as short summaries or not at all. Despite these drawbacks, the volume presents a valuable cross-section of research in the USA, especially (but not exclusively) in expert systems.

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.