Quaestiones Informaticae

Vol. 3 No. 1 February, 1984

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
Department of Computer Science and Information Systems
University of South Africa
P.O. Box 392, Pretoria 0001

Subscriptions

Annual subscriptions are as follows:

	SA	US	UK
Individuals	R6	$7	£3,0
Institutions	R12	$14	£6,0

Circulation manager
Mr E Anderssen
Department of Computer Science
Rand Afrikaanse Universiteit
P O Box 524
Johannesburg 2000
Tel.: (011) 726-5000

Quaestiones Informaticae is prepared for publication by Thomson Publications South Africa (Pty) Ltd for the Computer Society of South Africa and the South African Institute of Computer Scientists.

Program Design as the Tool of Preference in Computer Literacy Education, Experience, Incentives, Implications

J.M. Richfield
Shell South Africa

Abstract
Structured program design is urged as a rewarding vehicle for the introduction of computer concepts to age groups ranging from mid primary school to senior management. Progressively expanded subsets of the material form natural units after each of which the student may continue to more advanced levels, or stop with a logically coherent grasp of the subject so far. The intellectual and practical value of the concepts are emphasised together with the speed with which they can be instilled. Material requirements and training aims are illustrated with an outlined instructional procedure.

1. Introduction

After sociology, psychology, and economics, there are few fields of endeavour which spawn so much philosophical dissension as education. More than most contentious subjects, its product is both real and of crucial importance, but the scope for argument is varied and vast. Firstly, few practitioners of most subjects will agree on the best technical practice; few administrators agree on the selection of subjects or subject matter; few teachers agree on the best teaching techniques for a given subject; and no-one seems to agree on how to assess the results of the teaching. It has been quoted [1] that if computers had existed in the Middle Ages, programmers would have been burnt at the stake by other programmers for heresy. Teachers *did* burn other teachers at stake in those days, but the practice has largely been discontinued, to the apparent discontent of a large minority of the profession.

In view of this modern exemption from too-literal contribution to public illumination, I herein honour the venerable tradition of amateur contribution to the subject (if not necessarily to its advance). Not all the following opinions and observations are revolutionary and none are contaminated by excessive exposure to the theory of didactics; their justification relies on good intentions, strong views on the merits of the subject matter and varied experience in its teaching and application. This is no guarantee of soundness, but may serve to confuse hecklers. What I most dramatically lack is any pretense to controlled studies or other requirements for academic rigour — not out of disrespect for such, but simply for lack of the opportunity or incentive for the necessary investment of resources. You are invited to consider without prejudice the plausibility of the views and the promising empirical observations as grounds for formal investigation or practical application.

It is perhaps a bad sign that I shy away from too tight a definition of one of the key words in the title. Computer literacy — or computeracy, as I call it in analogy to numeracy — is a term which people airily apply to a number of concepts, ranging from confidence in use of a terminal without any idea of what is going on behind it, to a sophisticated appreciation of a wide range of technical subjects.

In the present context, fuzzy set theory is invoked to justify the use of the term for various levels vaguely between such extremes, and meeting certain criteria of competence. Knowledge of hardware mechanism is optional, as are computer language and data representation at the more modest levels. On the other hand, the ability to follow an algorithm in a familiar notation and the process of its generation are introduced practically at the start, together with a good idea of the consequences of the instructions occurring in the code. At the next level is an ability to apply certain disciplines to the generation of algorithms to solve a reasonable range of problems and to grasp in principle how more advanced problems might be approached. The ability to express algorithms in a programming language and to deal with implementation details follows, by which time we decidedly encroach on programmer territory. Simple computer literacy need not in principle be a sound basis for further study, but a student with the will and aptitude should be well placed to continue from any point in the process described here. If the promise of material and intellectual benefit held out unify the instructional professions to the extent of trying the approaches described here, I will be well rewarded.

2. Incentives

Like almost every practitioner of any subject, I regard mine as of special educational value and philosophical significance and the justification is not simply the growing importance of computers in the modern world. The generation and manipulation of abstract algorithms to achieve tangible results seems to me so fundamental in its effect on minds with any aptitude for such things, that the undoubted mundane value is of secondary interest. Its intellectual value I regard as rivalling pure mathematics, exceeding or rivalling that of any applied science and, as a skill in problem solving, of practical value ranking close after the three R's. So much for the first exhibit.

The next point is that it *is teachable,* at least to the calibre of pupil likely to select it as a subject. Teaching, no less than politics, is the art of the possible and it would be criminal to permit computer-related skills to be passed over on the grounds of unteachability. In fact, in the subject of algorithmic skills, it seems to me that the main differences between teaching children and adults arise firstly from the adults' greater repertoire of familiar skills and the perspective on problems and requirements lent by their wider experience. This eases the assimilation of certain classes of new skills and gives the instructor more scope in the selection of examples and exercises. On the other hand the children have their own strengths of greater retention of new facts and procedures as long as they have the necessary background knowledge and as long as they can immediately apply what they have learned. These provisos make certain demands of the instructor in the construction of examples and exercises, but they are not really restrictive. Also, perhaps because the children I have been in contact with have tended toward the bright side, they have shown themselves able to cope with abstraction rather better than most adults.

To me the two most exciting things about this approach are firstly that it is equally valid except in detail for any age group of student, from post-toddler to sub-decripit and secondly that the different levels of skill are more or less compatible. Most syllabuses suitable for the student aiming for a modest level of knowledge are unsuitable for the beginner of a professional-level curriculum. In extreme cases, even a fair amount of unlearning is in order, should the student change his course. However, in beginning with abstract structured design and justifying the process by demonstrating the fruits of the process on the computer, one can quickly supply the basic concepts and confidence required for the superficial student on the one hand, while on the other laying the groundwork for any future level of

algorithm, data, or procedure design required by the future professional. The conceptual structure is generalised and the smallness of the kernel of understanding required should have as much impact on the ease of teaching teachers as on teaching students.

Actually, in the right circumstances any keen student is likely to achieve more in the application of computers than one might predict from his rated academic level, if the work of such as Papert [2] is anything to go by. What the application of computer facilities does for computer literacy, is a subject in itself.

3. Obstacles

Perhaps because of its youth, computer and information science is rarely taught with any great technical competence in schools. All the best practitioners are too busy earning a living and sneering at the products of the educational institutions to contribute much to solving the problem. Teachers still tend to lack the in-depth experience necessary to appreciate the merits of those disciplines that have been shown to contribute to sound practice. Also, unexpectedly in view of the simplification introduced by the more popular of current techniques, it seems to demand a surprisingly advanced grasp of the principles to impart them with conviction. None the less, the simplicity is there, and enables one to produce in a short time and without tears, students who are house-trained and have a sound basis for further development.

The most maddening of the frustrations that beset anyone with views consistent with those below, is the combination of the obscene incompetence of design of language and support systems foisted on the market by the dominant vendors, with the smug idiocy of practitioners who see their ability to "get on with the job" irrespective of the tools, as a merit of those tools. This spurious merit in application they then tout as merit in education. Sometimes they see it as not mattering and sometimes as a positive virtue that the beginner be given a taste of things to come. Others point out with perfect truth that it is a bad language indeed that cannot be used in a style which conforms to the spirit of good structure and useful notation. That such expedients are aesthetically offensive may be an over-fastidious view, but they undoubtedly are severe sources of noise in the instruction of beginners.

The arguments that it is easy to be wise in hindsight and that the vendor without his own version of BASIC together with his own horrible extensions will go to the wall, hold no water. The hindsight has been accessible to all for perhaps fifteen years, but the unifying feature of almost every BASIC to come on the market is the ingenuity with which their language extensions ape each other without achieving compatibility, generality, elegance, or even usability. The stench of ad hockery is pervasive and it is hard to believe that we have anything but the whims of systems programmers to thank for the motley collections of rarely-used instructions which clutter the manuals. The largely counter-productive obsession of the vendors with locking-in their customers seems to inhibit their support of the recognised structures. This conflicts with the stated justification of BASIC as the universal language.

The design of languages and extensions by the academic community has not been free of acrimony or monstrosity, but their most prominent prospects for general introduction have all been usable. Meanwhile, the main difficulty in supplying students with sound tools to exploit the exciting equipment now available, is that most of the cheapest and most attractive either support none of the acceptable notations, or do so unacceptably clumsily, or as very expensive options. This perpetuates the situation as a sort of technological-commercial blackmail. The simple production of better rools may be feasible for the individual user, but the effort is sizable even if he is equipped for the task; the delay in implementation of the teaching project is often prohibitive; and the fact that he is then saddled with the maintenance of an incompatible package on an obsolescent system is a powerful disincentive to independent action. This does not prevent sound education, but it severely inhibits it, increases the expense and persistently introduces contamination from proponents of the quick-and-dirty.

4. Desiderata

What would be the features I would specify in a system to suit the teaching methods I recommend? Actually the requirements are rather simple and even loose. As a practitioner predominantly of functional decomposition using pseudocode, I want a target language which is usable as pseudocode with minimal modification of syntax. In particular, violation of functional structure in translation to the target language is undesirable. It is also a nuisance if it requires too cumbersome a superstructure of declaratives. Declaratives are tolerable at worst, but it makes life easier if they can be neat and compatible in concept and format, with the procedural text. It is also nice if it is convenient to omit them from the students' first efforts, or at the least to invoke them from a library or some other automatic source. Unusually among evangelists, I do not necessarily consign other notations and methods to outer darkness, but do insist that the road to salvation lies through the ability to handle abstraction at varying levels, both bottom up and top down, without contamination from unstructured entry or exit.

The support system should be fairly user-friendly, but can be obscure if that is the price of brevity and convenience. It is less disruptive to be told to enter a string of say two letters, a delimiter and carriage return when you start, a similar but different one to run, and another variant or two to stop etc, than to understand the helpful technical terms which appear, variously mangled, in typical commercially-oriented command languages.

A natural language command handler is nice in theory, but most implementers seem to think that they achieve such a facility by using long english words for mnemonics, arbitrarily selected, often partly abbreviated and with rigid American spelling. In practice most naive users seem happier with few, short commands, even if they do not understand the mnemonics, as long as access to a menu or list is instant and convenient. It is also important that the circumstances for the use of each are easily described and recognised. Children in particular take to this like ducks to water; in fact, they sometimes revel in the esoteric terms, apparently relishing them as "in" words. They also like single-stroke function and abbreviation keys and complain (but quickly adapt) when new systems lack accustomed features.

Another attractive approach is to accept full English words but to require only enough letters to avoid ambiguity in context. The minimisation of keystrokes and of independent concepts appear to be key factors in achieving the kind of user friendliness which appeals to the budding computer initiate. A more worthwhile aim than a misleading resemblance to English is that the work required to identify and specify an action is small and disaster does not lurk in a thoughless keystroke. It should not be possible to destroy the system or the last session's work, or even find oneself bootstrapped out of the current game without elaborate idocy. Automatic saving of one's status at helpful points is also nice if available.

It is more important to ensure speed and reliability of response and convenience of job, data, and code submission, editing and access than to make the commands English. This makes interactive facilities attractive and local intelligence preferable to remote computer processors. Local time sharing with several access points connected to a processor in the same room is all right, as it not only cuts out the communications unreliability, but makes it clear what happens when sharing. In particular, if the system goes down, all can swear or groan together, without the bitterness, frustration and confusion of wondering whether the remote terminal has gone glassy-eyed because of one's own fault, line-, terminal-, processor-, software-, or systems-programmer-fault. In fact the committed pupil will accept monthly response by runner with cleft stick, but as a rule this takes the fun out of things. No fun — no learn. Still, although the modern child is too blasè to put up with the systems regarded

as miraculous even a generation ago, one *can* run a successful course with unimpressive resources.

In summary, happiness in a school environment would be a system of personal computers, probably connected, possibly with a few screens each, supporting at least one suitable language, say Modula 2, preferably in both interactive and compiled versions, and with a variety of I/O devices, including modest-resolution interactive graphics and high resolution hard-copy graphics. Data storage should not be a restriction in speed or volume.

It is important to get the requirements into perspective. One cannot expect to get the best results in strongly unfavourable conditions. In particular, if the students cannot get at some facilities to see the results of their efforts, it would be naive to expect most of them to maintain much enthusiasm. At the same time, the methods outlined here do not compare unfavourably with any others under matching restrictions. Much can be done with chalk and in realistic circumstances one must expect computer access to be in short supply. The compromises one must then make may be disappointing, but need not always be disastrous.

5. Generalities

There are many valid routes to a working knowledge of computer related matters. Like most practitioners of my generation, I began on second-generation machines in low-level languages and developed from there. It suited me, and my only regret is that structured concepts were not in general currency in those days. For years my prescription for the beginner was to begin at the low level and build on the sound foundation thus provided. Since then several things have happened to reduce the general applicability of that advice. For one thing, it is now commonplace to require a non-trivial understanding of matters concerning computers without being in the DP industry as user or vendor etc. Secondly, it is the rule rather than the exception to use high level interfaces only. An amazing number of commercial programmers nowadays do not even know what a low level language is. The thought is somehow disgusting, but the material grounds for objection are few — and shrinking.

Now, apart from the question of the academic soundness of the run-of-the-mill practitioner and the emergence of those peripherally associated with DP, another class of educational requirement has emerged. Children started the invasion as scattered secondary-school students, but they now bid fair to become the major consumers of all computer-related education. What is more, it now seems to me that their computer education should overlap the three R's and precede sexual instruction. And in this case there are at least two factors invalidating the low-level to high-level approach as a general policy. Firstly, the intellectual relevance of the technical low-level information is far lower. Secondly, while some of that type of material is easily assimilable and even interesting to some children, it is a large subject and few would have an appetite for enough to constitute a course, even if it were of enough practical relevance or urgency to justify the investment in school facilities.

On the other hand, the basics of program design as an abstract subject can nowadays be imparted in a few hours of classroom time and if suitable computer facilities are available, can yield pleasing and rapid concrete results; it forms the foundation of a subject wide in its own range, rich in intellectual content and of value as a mental skill and practical tool in many other subjects. Such a combination of teachability and teachworthiness is surely the educator's dream. The youngest children I have tried it on were six years old. To the limits of their arithmetic abilities, they did rather well at it. Eight-year-olds were easier to deal with, but I should hesitate to undertake to teach it on a routine formal basis to such an age group. Keen, bright ten-year-olds are no problem. One must of course distinguish between instruction of individuals and the classroom situation. It is hard to see how one could go about teaching a mixed-ability class of fair size and varied motivational level, but as an elective subject, it is a beauty.

Of course, other approaches enables one to impart non-trivial computer skills to even younger children. Most prominently perhaps, Papert [2] has impressively demonstrated the development of both programming and debugging in his pupils. He argues that both are skills of many unexpected intellectual benefits and that each is worthy of recognition in its own right. I have no fault to find with these views, but think that it will be some time before we can apply them on a large scale. Also, the subject matter I recommend is for the most part more rigidly structured — more of a discipline. This is not to denigrate unstraight-jacketed education, or even to deny its legitimate place alongside formal material in the same subject, but for the age groups in question the formal material is accessible and can yield rapid, predictable results. Besides, taught as a discipline, these skills are more easily applicable to recognisable cases and the procedures are more calculated to avoid wasteful false starts. These are features attractive in budgeting curriculum time. There is no conflict between the approaches; they are compatible in concept and in practice. In fact a fair percentage of the material I prefer to present in an exploratory fashion, using the computer as a tool; and preferably a visual, interactive tool.

6. Key points in the procedure

The exact approach and sequence may vary according to taste and especially according to the facilities available. The following text assumes a rather generously equipped class, with at least one interactive screen per small group.

6.1 Introductory remarks

There is not much special about this. The usual considerations apply for establishing rapport, requirements, expectations etc. It is perhaps more important than usual to put the class at ease and to ensure that there is no reluctance to ask questions, as the process is futile if not participative.

6.2 Basic concepts of process

The best way to illustrate this is to demonstrate a very simple program in an interactive interpretive language, very transparently written with descriptive vernacular variable names and with *NO COMMENTS*. Comments are apt to be taken for part of the functional code. There is plenty of time to introduce them a little later — at this point they are so much noise.

Whatever examples are shown should be impeccably structured and religiously adherent to good algorithmic practice — pre-reading before loops and all that. *BUT* this is not the time to point it out to the innocents; let them simply think that is the way one does it. Do not evade pointed questions, however, but even then, do not elaborate. Simply explain that violations of taboos cause hair to grow on the palms and that all will be explained anon — and do not forget to explain it anon!

To illustrate: a good starter is something like a counting program which chattily promps for a digit of input then prints that many numbers from one up in sequence. Let the students satisfy themselves as to the range of function of the program — usually a minute for adolescents and up, but tots will sometimes raise the roof if interrupted too soon. Be that as it may, before boredom threatens, but after the first flush of enthusiasm, one shows them how to list the program, then takes them step by step through it, not explaining anything unless absolutely essential for *immediate* understanding. The purpose at this point is to associate each statement with what was seen on the screen, emphasising the sequence of events and changes of value. As soon as this has been done, however sketchily, encourage the students to experiment — first with modifications to the text, say changing prompts, annotation literals and count constants or whatever, then with changes to statement sequence and duplication or omission of statements; and lastly with enhancements, such as putting the whole code into a loop so that it repeats the whole rigmarole after each performance.

A cardinal danger is the temptation to explain too much. There are dozens of things one can explain, such as the significance of the commands to run the program or editing functions, or the syntactual rules of the language, but now is

not the time. what seems to work perfectly well without confusion, jeopardy of future subject matter, or student dissatisfaction, is simply to say that one does so-and-so at this point to do such-and-such and, if importuned, to promise that all will be revealed in good time, but that at the moment we want to carry on.

It is amazing how many foundations for further conceptual development are laid at this point, but it is also now that one first encounters the fact that only the exceptional student has done more. All one can rely on is receptivity for the next stage, and even that evaporates if not quickly reinforced, preferably by another and slightly more advanced program example. One I have used with success prompts for a number and moves an asterisk across the screen that many times. It then repeats the process until an end of job condition is encountered. This introduces string handling through changing the asterisk's "tracks" and various other whims.

About now is a good time to consolidate what has been achieved by identifying and naming the key concepts, so back to class. The first thing *I* do at this point, is to teach the Dijkstra structures. The notation used to illustrate the process flow is old-fashioned flow-charting, using only the rectangular process block and the diamond-shaped decision block. Obviously the choice of notation is not critical, as it is the only exposure to flowcharting that they get from me. All that matters is that it must be very simple and very visual.

The point of using flowcharts at this stage is that what is being illustrated is just that: FLOW; the flow of control in the sequence, selection and iteration blocks. This flow is not explicit in the notation of the typical block-structured language and explanation that the bottom of a loop means that execution continues from the top until termination, is troublesome to many students. Starting from the sequence construct also permits flowcharts to dramatise the occurrence of such components in loops and selections; the once in, all the way through and once out nature of each construct; and the consequent naturalness of nesting blocks of anonymous content to indefinite depth. As the concepts are simple and easily memorised, flowcharts are not needed afterwards. Once the block structured notation is understood, the class is generally perfectly happy to use it, suitably indented, as program code and documentation, as it is now as visual, less trouble to code, and directly compatible with the available I/O media. I have yet to hear a pupil complain on being told that flowcharts are from now on to be ignored.

Incidentally, one of the few things I insist on having memorised, is the sequence-selection-iteration triplet. Terminology can be a vast help to fixing and accessing concepts in perspective, and this is one case where it works miracles.

Throughout the discussion, there is continual reference to the code they have run, to identify rather than merely recognise the structures they are learning. Further examples of very small illustrative segments of code can be elicited in this phase of the class. How much more machine time is to be consumed in the process depends on available facilities and on requirements.

The problem of termination of loops due to circumstances arising halfway through their code is ideally dealt with by demonstrating the venerable nested-if-and-flag-variable technique. Having used it till the problem and the structure are ingrained, one can graduate to the loop exit statement and stay with it, as long as the available software supports it. The goto remains a nono.

6.3 Concepts of data and data manipulation

So far the concept of data has probably been treated very casually. Perhaps the instructor has even managed to avoid talking of constants and variables, or if not, to sidestep the question of storage. Notice that everything has been very abstract and none the worse for it. The primary reason for leaving things out so far, has been that there is so much to tell, but there is an interesting by-product. Since everything has been structured in coherent modules of concepts, one can let the student who has mastered all he needs drop out of the course at any of the frequent points when a subject has been buttoned up for the moment. What he never asks is largely what he never needs if advanced study is not his requirement.

However, this is about as far as it is comfortable to go on without considering data in greater depth. The first thing is to tell as much as is necessary about the nature of data storage in the machine. The approach depends very little on the background of the class. Even if they know about such things, binary, floating point and the like are not at moment of the essence. The main points are that the storage is finite and that the entities are pigeon-hole-like and can be labelled and accessed as either constants or variables. This need hardly be illustrated with new code, as the examples used so far are ample.

The assignment statement merits explicit attention at this stage, in combination with expressions. Using the pigeon-hole illustration, one can easily show the sequence of operations needed to achieve the effects observed in the running of the programs, including temporary storage locations etc. Here one also can deal with the hierarchy of operators. At points like this one can generally carry on at a moderate speed with advanced ten-year-olds, but the younger children often have too little background. They can pick it up very quickly in this context, but that is hardly the primary object of the class. Partly for this reason and partly for fear of clashing with the educational authorities, it may be better to limit oneself to very simple expressions.

Now comes a major step: the introduction of subscripting. It is not difficult to illustrate, pigeon-hole style, but it is the key to the next set of live examples. The important points are the array membership name, the use of arbitrary expressions as subscripts and the possibility of an arbitrary number of dimensions. Suitable subjects for illustrative programs are sorts, searches, and two-dimensional placing of characters on the screen. Incidentally, this also makes for a very easy introduction to graphs and functions.

By this time it would labour the point to detail the steps. Suffice to say that more sophisticated data structures such as lists, stacks, trees etc, derive easily from the same process. The functional decomposition approach developed accomodates the generation and modularisation of code naturally and the sky is now the limit. No common school of disciplined development is excluded, data flow, data entity, functional, the lot. With a little trouble one could probably explain what a go to is, though explaining why one might want to use it could present problems.

7. Class timetable

Of course, the speed of progress will vary with circumstances, but I hope it is clear that by any standards the rate is very rapid. Really advanced concepts become accessable within a day or so of class time if the facilities are available. If what was once regarded as essential background is never required, it may be ignored. Otherwise it is now easily and quickly assimilable in a context of well-understood goal-directedness of programming.

However, that was the good news. If that day of instruction is not rapidly followed by more of the same, it trickles out in no time. It leaves behind an impressive receptivity to related material, but not enough to repay the effort and loss of momentum. Also, the younger students show a curious tendency to forget, not isolated facts and concepts, but integrated algorithms. Until they have used several of them several times, the most gratifying receptivity and even creativity in class is so much fairy gold. What is required is something like a week's concentrated course. This may not sound much, but in eight-hour days it amounts to some eighty classroom periods or nearly a semester at one period per day. This can not generally be demanded in term time, but if the necessary facilities can be located and an appropriately sized group of enthusiasts can be assembled, much should be possible.

8. Parthian shot

One of the most tempting thoughts to arise from the contemplation of a supply of youngsters with a sound basis in design

at the beginning of the curriculum, is the prospect of teaching them sound follow-up material. The typical matriculant with computer science has as much to unlearn as to learn at university level, and one can say as much for the typical graduate entering industry. Sound applied practice is in principle no harder to teach than unsound, and a good deal more rewarding. Unfortunately it is less cohesive and less well-known, which is saying a lot. The problem is three-fold: to teach the children; to teach the teachers; and to teach the constructors of syllabi; probably in ascending order of difficulty.

References

[1] D. W. Barron, Recursive Techniques in Programming. London, Macdonald 1968, (quoting Gill, S. 1960).

[2] S. Papert, Mindstorms: Computers and Powerful Ideas. Brighton. Harvester Press 1980.

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:

Department of Computer Science
University of South Africa
P.O. Box 392
Pretoria 0001
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 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.

Quaestiones Informaticae

Contents/Inhoud

*Presented at the Third South African Computer Symposium held on 14th to 16th September, 1983.

LITHO