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.

RESEARCH DIRECTIONS IN INFORMATION SYSTEMS

J. Mende

Department of Accounting
University of the Witwatersrand, Johannesburg, 2001

ABSTRACT

The success of a research project depends upon the inputs and outputs selected by the researcher, a poor choice entailing useless results or wasted efforts.To avoid this one might consider the following alternatives:

1. Our present knowledge in the subject Information Systems largely consists of facts and engineering techniques, scientific principles being few and inadequate. This imbalance suggests that research aimed at extending the *scientific foundations* of the subject should yield more significant results than purely technical research.
2. Structural and functional similarities between information, business and academic systems indicate the existence of general laws that apply accross the interdisciplinary boundaries of the corresponding subject areas. Accordingly it should be possible to *transfer* principles and techniques from Business and Education to Information Systems and vice versa.
3. Economic History predicts that mankind is moving towards an Edensqùe state in which no one is obliged to work for a living. The information systems necessary in *that state* therefore constitute highly significant problems for research.

THE FOUNTAINHEAD OF PROGRESS

In the dawn of history primitive Man struggled for survival in an inhospitable environment, condemned to toil from dawn to dusk simply to secure the bare necessities of life. "You shall gain your bread by the sweat of your brow" (Genesis, 1974:4)

However, over the ages people made special efforts, beyond the minimum required for subsistence, to establish "fixed capital" - non-consumable machines, systems and education - which facilitate the subsequent production of consumables such as food, clothing, etc. As a result of those efforts, today we achieve a substantially higher standard of living with considerably less effort than our ancestors. And if the process of capital formation continues, Man can aspire one day to evade Adam's curse altogether and satisfy his creature needs virtually without any labour at all (Hohenburg, 1968:3).

Capital formation involves three distinct activities (Mende, 1985):

- routine production of machines, systems and education
- development of the designs of these items and their productive processes
- research to provide a basis for design.

These vary in their potential for diminishing the ratio of work to consumption. Present-day capital items still demand a large proportion of human involvement in their operation, and therefore producing more of them can only marginally diminish the ratio. To effect sizeable reductions demands dramatically improved designs, and so development activities offer much greater opportunities for progress.

However, design is limited by our - still very incomplete - understanding of Nature and Man, and truly revolutionary designs require corresponding advances in the natural and human sciences. Consequently *research* is the most significant of all the capital formation activities: the very fountainhead of progress towards Man's release from Adam's curse.

THE PROCESS OF RESEARCH

Accordingly universities, research institutes and the R & D departments of business corporations throughout the world employ people to carry out the research that is crucial to progress. Like many others, the process of research requires inputs, methods and outputs, success depending on the optimal choice of these three components.

The *output* consists of new knowledge - facts, principles and techniques that were previously unknown. Its significance hinges on the resulting benefits to mankind, in particular the designs of new capital items which can be derived from it: if one selects an aspect of information systems which urgently requires improvement, the chances of a significant research outcome are greater than if one selects an aspect which is quite satisfactory. Consequently to ensure success the researcher should carefully select the problem to be solved, choosing the one whose solution promises maximum benefits to mankind.

Similarily success depends upon careful choice of research *inputs* - empirical observations, readings in the subject literature, ideas in related subjects, original thought, etc. If one chooses an inappropriate input - for example empirical observation when the result can be derived by deduction from established theory, the research may require much more effort than really necessary.

The same applies to the choice of *method* - statistical hypothesis testing, mathematical modelling, logical inference, etc. An inappropriate choice of method can result in outright failure, for example a statistically insignificant result, or an inordinately long project duration.

Consequently if one leaps into a research project without careful preliminary selection of the most appropriate outputs, methods and inputs, a large proportion of the research effort may be wasted. To avoid this risk, one ought to consider various *alternatives* at the commencement of the project.

In identifying such alternatives one may consult existing literature. For example, textbooks on Research Methodology (e.g. Labovitz & Hagedorn, 1976) recommended various methods and inputs; and the " further research" sections of doctoral theses suggest potential outputs. The remainder of this paper presents some further alternatives - the author's favourites - which are particularily applicable in the field of Information Systems.

THE NEED FOR SCIENTIFIC PRINCIPLES

Firstly, it is suggested that the researcher should consider producing outputs in the form of scientific principles (Toulmin, 1953) - concepts and statements of specific relationships that exist between them - as typified by Ohm's Law: "current is proportional to voltage".

Analysis of established subjects such as Physics and Medicine reveals the existence of three distinct kinds of knowledge:

- facts, e.g. the fuction of the heart, the radioactivity of uranium
- principles, e.g. Archimedes' Principle, Ohm's Law of Resistance
- techniques, e.g. open heart surgery, the nuclear reactor.

These are complementary in alleviating the human condition: engineering techniques are based on scientific principles, and principles are based on facts.

Classifying the field of knowledge *Information Systems*, we find for example that the topic:

- Hardware is largely factual, e.g. functions of the disc
- Software is largely factual, e.g. fuctions of IF, sort, IMS
- Programming is largely technical, e.g. structure charting
- System Design is largely technical, e.g. flowcharting
- System Analysis is largely technical, e.g. data flow charting
- Information Systems planning is largely technical, e.g. top-down and bottom-up strategies.

Scientific *principles* are few and far between, e.g.

- a program consists of sequences, selections and iterations (Jackson, 1975:16)
- highly coupled modules are difficult to maintain (Yourdon & Constantine, 1979:85)

Comparing the relatively primitive and failure-prone application systems developed by information systems practioners with the highly sophisticated and succesful hardware devised by our colleagues the electrical engineers, one wonders what causes the disparity in achievement. An explanation of this is that the electrical engineers have much more extensive foundations for their engineering techniques, i.e. the science of Physics. This hypothesis is supported by the success of other subjects such as chemical engineering - based on well developed foundations of Chemistry - and Medicine, which relies heavily on principles of Biology, Biochemistry and Biophysics.

Further evidence is furnished by a classical law of Economics, that if a process employs multiple inputs, their marginal products should be equal. Applying this to the I.S. profession which employs inputs of factual plus scientific and technical knowledge to produce an output of practical systems, a marked deficiency in scientific principles would result in sub-optimal achievement. A similar effect is predicted for the process of research on Information Systems techniques.

Therefore it is to be expected that the lack of science in Information Systems could distinctly impede practical systems development as well as academic systems engineering research. Consequently research outputs that help establish a broad base of *information systems principles* are likely to prove highly significant.

PARALLELS WITH RELATED SUBJECTS

Secondly, there is reason to expect that researchers in the subject Information systems could obtain useful inputs from *related subjects* such as Business and Education.

Simpler information systems typically consist of two subsystems (Mende, 1982):

- a collection subsystem, comprising programs which collect data from sources in the system's environment and store it in a set of files
- an extraction subsystem, comprising programs which extract information from the files and distribute it to the users in the system's environment.

These are similar in function to the two major components of a trading firm:

- purchasing, comprising people who collect goods from suppliers in the firm's environment and store them in a warehouse
- marketing, comprising people who extract goods from the warehouse and distribute them to customers in the firm's environment.

And they in turn are also similar in function to the two major subsystems of academe:

- research, which collects knowledge about the world and stores it in libraries
- teaching, which extracts knowledge from libraries and disseminates it to students.

These functional and structural parallels may well represent particular instances of a *general body of principles* which apply across inter-disciplinary boundaries of Information Systems, Business and Education. General System Theory (Von Bertalanffy, 1972:14) prophesies the existence of many such principles, which "manifest themselves as analogies or logical homologies of laws that are formally identical but pertain to quite different phenomena or even appear in different disciplines". Consequently researchers in Information Systems should consider examining other subjects for potential inputs. And vice versa, they might look for opportunities of transferring Information Systems principles to other subjects. For example, the author has transferred Present Value ideas from Finance to Information Systems (Mende, 1984), and success criteria from Information Systems to Education (Mende, 1981).

FUTURISTIC RESEARCH

Finally The Eden Ideal suggests the need for a "futuristic" as distinct from an "evolutionary" approach to research.

Traditionally research is viewed as a cumulative process of adding new items to the growing stockpile of mans knowledge (Kuhn, 1970:2), each individual researcher building upon a foundation laid by his predecessors. The History of Science has modified that view on empirical grounds, replacing it with a perspective of knowledge successively evolving through turbulent phases of theoretical reconstruction and factual re-interpretation followed by calmer periods of cumulative expansion. Now the Eden Ideal challenges both the previous views on normative grounds.

If Man is ever to attain a state of no obligatory work, then in every decade researchers should provide a knowledge base for the new designs to be developed in the next. Consequently we need a future-oriented view of research, not as a process of extending or reconsructing past knowledge, but as a *deliberate preparation for the future.*

This has subtle implications for the researcher in the field of Information Systems. Rather than examine existing systems, he should investigate those required in Eden; instead of aiming to perfect the theories or methods of the past, he should seek those required to restore Eden. His attention should be firmly directed towards the information systems of the future.

That demands some pre-knowledge of those systems: finer detail in the broad framework of Eden as a state without obligatory work which suggest specific research inputs and outputs.

Such forecasts are obtainable by determining the characteristics of machines and supporting information systems which follow from the premise of no human involvement in their operation, coordination, manufacture or maintenance. For example, suppose a factory's machines are to be repaired without human intervention. Then every machine would need to be equipped with sensors which allow a supporting information system to detect malfunctions and despatch maintenance robots to replace defective parts. This hypothetical information system constitutes an input to the research process, and its problems suggest potential outputs.

As the solutions to such problems are likely to represent "breakthroughs", the researcher may well achieve enormous success by selecting inputs and outputs based upon a *forecast* of the information systems required in Eden.

PARALLELS WITH RELATED SUBJECTS

Secondly, there is reason to expect that researchers in the subject Information systems could obtain useful inputs from *related subjects* such as Business and Education.

Simpler information systems typically consist of two subsystems (Mende, 1982):

- a collection subsystem, comprising programs which collect data from sources in the system's environment and store it in a set of files
- an extraction subsystem, comprising programs which extract information from the files and distribute it to the users in the system's environment.

These are similar in function to the two major components of a trading firm:

- purchasing, comprising people who collect goods from suppliers in the firm's environment and store them in a warehouse
- marketing, comprising people who extract goods from the warehouse and distribute them to customers in the firm's environment.

And they in turn are also similar in function to the two major subsystems of academe:

- research, which collects knowledge about the world and stores it in libraries
- teaching, which extracts knowledge from libraries and disseminates it to students.

These functional and structural parallels may well represent particular instances of a *general body of principles* which apply across inter-disciplinary boundaries of Information Systems, Business and Education. General System Theory (Von Bertalanffy, 1972:14) prophesies the existence of many such principles, which "manifest themselves as analogies or logical homologies of laws that are formally identical but pertain to quite different phenomena or even appear in different disciplines". Consequently researchers in Information Systems should consider examining other subjects for potential inputs. And vice versa, they might look for opportunities of transferring Information Systems principles to other subjects. For example, the author has transferred Present Value ideas from Finance to Information Systems (Mende, 1984), and success criteria from Information Systems to Education (Mende, 1981).

FUTURISTIC RESEARCH

Finally The Eden Ideal suggests the need for a "futuristic" as distinct from an "evolutionary" approach to research.

Traditionally research is viewed as a cumulative process of adding new items to the growing stockpile of mans knowledge (Kuhn, 1970:2), each individual researcher building upon a foundation laid by his predecessors. The History of Science has modified that view on empirical grounds, replacing it with a perspective of knowledge successively evolving through turbulent phases of theoretical reconstruction and factual re-interpretation followed by calmer periods of cumulative expansion. Now the Eden Ideal challenges both the previous views on normative grounds.

If Man is ever to attain a state of no obligatory work, then in every decade researchers should provide a knowledge base for the new designs to be developed in the next. Consequently we need a future-oriented view of research, not as a process of extending or reconsructing past knowledge, but as a *deliberate preparation for the future.*

This has subtle implications for the researcher in the field of Information Systems. Rather than examine existing systems, he should investigate those required in Eden; instead of aiming to perfect the theories or methods of the past, he should seek those required to restore Eden. His attention should be firmly directed towards the information systems of the future.

That demands some pre-knowledge of those systems: finer detail in the broad framework of Eden as a state without obligatory work which suggest specific research inputs and outputs.

Such forecasts are obtainable by determining the characteristics of machines and supporting information systems which follow from the premise of no human involvement in their operation, coordination, manufacture or maintenance. For example, suppose a factory's machines are to be repaired without human intervention. Then every machine would need to be equipped with sensors which allow a supporting information system to detect malfunctions and despatch maintenance robots to replace defective parts. This hypothetical information system constitutes an input to the research process, and its problems suggest potential outputs.

As the solutions to such problems are likely to represent "breakthroughs", the researcher may well achieve enormous success by selecting inputs and outputs based upon a *forecast* of the information systems required in Eden.

CONCLUSION

Accordingly when selecting the inputs to a research project, one might consider

- searching other subjects for tranferable ideas
- predicting the information systems requirements of Eden

And when setting research objectives - i.e. the results to be otained from a project - one might choose to

- establish scientific principles
- solve problems relating to Eden's information systems.

REFERENCES

Genesis, 1974. The New English Bible. Harmondsworth, Middlesex: Penguin Books, 1777 p.

Hohenburg, P. 1968. A Primer on the Economic History of Europe. New York: Random House, 241 p.

Jackson, M.A. 1975. Principles of Program Design. London: Academic Press, 299 p.

Kuhn, T. 1970. The Structure of the Scientific Revolutions. Chicago: University of Chicago Press, 210 p.

Labovitz, S. & Hagedorn,R. 1976. Introduction to Social Research. New York: McGraw-Hill, 147 p.

Mende,J. 1981. Developing Instructional Systems. The Commerce Teacher, vol. 13, 10-25.

Mende,J. 1982. Teach Systems the Deductive Way. The Commerce Teacher, vol. 14, 43-45.

Mende,J. 1984. A viability criterion for computer system development projects. S. Afr. J. Bus. Mgmt., vol 15, 144-149

Mende,J. 1985. Academic Keys to Eden. Johannesburg: working paper, University of the Witwatersrand.

Toulmin, S. 1953. The Philosophy of Science. London: Hutchinson, 176 p.

Von Bertalanffy, L. 1972. The History and Status of General Systems Theory in ' Systems Analysis Techniques' by Couger & Knapp, 1974, New York: Wiley, 509 p.

Yourdon, E. & Constantine,L. 1979. Structured Design. Englewood Cliffs, New Jersey: Prentice Hall, 473 p.

The SECD Machine : An Introduction

ROBERT DEMPSTER
Department of Computer Science
University of Natal, Pietermaritzburg
3200

INTRODUCTION

The S(tack), E(nvironment), C(ontrol) and D(ump) machine was invented by Landin (1964) to perform the mechanical evaluation of symbolic expressions. The evaluation of an expression invariably involves the application of an operator to an operand. The concept of an applicative expression (AE) is thus formally introduced and discussed. This is followed by an overview of the SECD machine and a detailed trace of the SECD machine state transformations during the evaluation of an AE on the SECD machine.

APPLICATIVE EXPRESSIONS

Landin (1964) defines an AE as either

- an identifier
- or a λ-expression (λexp) which has a bound variable (bv) which is an identifier or identifier list, and a λ-body (body) which is an AE,
- or a combination which has an operator (rator) which is an AE, and an operand (rand) which is an AE.

Identifiers comprise single- and multi-character constants and variables. The decimal numbers are also considered to be identifiers and as such they represent single entities viz. the value of the decimal numbers.

Examples of identifiers are trivial except that, as will be seen later, such identifiers are bound to values within a certain environment. As such they may have simple values or may in turn represent AEs.

λ-expressions can be illustrated by drawing parallels with conventional high school mathematical functions, e.g

$$f(x) = x^2 + 2 \qquad f = \lambda x.x^2 + 2$$

$$g(x,y) = x^2 + y^2 \qquad g = \lambda(x,y).x^2 + y^2$$

The notation used places the bv part between the λ and the dot which is immediately followed by the λ-body.

In the context of a computation AEs may appear in the following forms.

As an identifier the AE represents some value which will be determined by the environment in which the identifier is bound. This value may be a primitive i.e. a constant of the environment or another AE.

As a lambda expression the AE represents functional abstraction. This form of AE will always denote something provided we know the value of each variable that occurs free in it. For example

$$\lambda x.x^2 + y^2 + 2$$

will be valid provided the free variables y and 2 can be bound to specific values. It is worth noting here, that "2" is treated as the numeral "2" bound in some environment to the integer 2. Also that other identifiers such as square, sin etc. may exist bound in the same environment to basic functions which are effectively applied in one step.

Thirdly the AE expression may occur in the form of a functional application or operator/operand combination. Here both the operator (rator) and operand (rand) are in turn AEs and as such the rator and rand will have to be evaluated before the former can be applied to the latter. It is worth noting that mathematics makes no assumptions regarding the order in which we evaluate the rator and rand of a functional application.

To make sense the rator, if it is not an identifier bound to some function, must then be an AE which will evaluate to a function. The rand must either be an identifier bound to a value within the domain of the rator or it must be an AE which will evaluate to some such value.

From this we will gather that AEs are a type of composite information structure. In order to process such information structures, we require various operations. Landin (1964) suggests that we will require predicates for testing for alternative AE formats as well as selectors and constructors for partitioning and building AEs respectively. Typical examples are:

```
is-λexp (λx.x + 2) = true

sel-bv (λx.x + 2) = x

constr-λexp (sel-bv(X), sel-body(X)) = X
```

where X is a λ-expression.

In the earlier definition of AEs the bv of a λ-expression was described as an identifier or an identifier list. The texts referenced for this paper tended to use λ-expressions with a single bound variable. This is because it is possible to replace a function of several variables by a complex function of only one variable, e.g.

```
λ(x,y).x + y   →   λx.λy.x + y
```

Although more complex the latter representation is more suited to the mechanical evaluation process as performed by the SECD machine.

AE EVALUATION

To be useful in the context of computing, an AE will ultimately have to take on some value. This value may either be a number or a function or a list of numbers or functions. In order to evaluate an AE we will have to be able to determine the value of any variables occurring free in the AE. Thus we must associate with the AE an environment (E) which contains the necessary free variables appropriately bound. This environment is readily implemented in terms of a structure consisting of name/value pairs. We thus associate with the evaluation of an AE a closure which consists of the AE and its appropriate environment. The construction operation constr-clos(λexp, E) does just that while is-clos is a predicate for testing for the presence of a closure. The selectors clos-body, clos-bv and clos-env select the respective parts of the closure in question.

STACKS

As stacks are crucial to an appreciation of the SECD machine a brief discussion of the stack structure and operations employed follows.

The stacks are implemented as linked lists and can thus be processed using the usual list and stack operations. Consider an empty or null stack S = []. After the push operations push (z, S), push(y, S) and push(x,S) we have S = [x, y, z] with the element x on the top of the stack. A notational device '::' is used to denote the result of a push operation and in this context push(w, S) will result in a new stack value represented by w :: S i.e. [w, x, y, z].

In terms of the linked lists underlying the stacks we also employ the selectors head (h) which selects the element on the top of the stack, tail (t) which effectively selects the popped stack and then 1st, 2nd etc. which select the first, second etc. elements from the stack respectively.

Using the stack S described above

```
h      S  =  w
t      S  =  [x, y, z]
1st    S  =  w
2nd    S  =  x  etc.
```

A final note regarding the stacks is that as the stack elements are variant they must thus be tagged in order to allow the type of any element in question to be determined. Suitable predicates are employed for this purpose.

THE SECD MACHINE

Like all other computing machines the SECD machine proceeds through a number of states as it evaluates an AE. The SECD machine state is represented by a 4-tuple of stacks i.e. <S, E, C, D>. S is an evaluation stack which is used for computation in terms of the application of a basic function on the top of the stack to an operand(s) immediately beneath it. The result replaces the operator and operand(s) on the top of the stack. E is a stack used to store the current environment. It consists of a number of name/value pairs and can also be considered as an association list. C is the control stack and it initially contains the AE which is to be evaluated. D is the dump and it is used to store the current state of the SECD machine each time a nested function application takes place. This is done by pushing the current state onto D and then initializing S, E, and C in terms of the nested function application. After the nested function application is complete the previous SECD state is restored by popping it off D.

The SECD machine is conceptually not unlike conventional von Neumann machines in the context of the latter's execution of programs written in procedural languages such as PASCAL. When the code for these programs is executed a stack frame is created on the stack for each procedure evocation. The stack frame is typically built up from the address to which control must return after the procedure terminates, the procedure parameter list, and the list of variables declared local to the procedure. When the procedure terminates the stack frame is removed from the stack exposing the stack frame associated with the calling procedure. If the procedure was typed, (i.e. a function) then the value returned by the function would have been left on the top of the stack. The dynamic nesting of procedure calls is thus captured on the stack. This situation does not however necessarily reflect the static nesting or scope of the currently evoked procedure. To keep track of the static environment of the currently evoked procedure a set of display registers is used. These describe the static environment by means of pointers to the relevant stack frames and are used to resolve the visibility of variables within this environment.

In this context the 'stack' of the SECD machine is the current stack frame, the 'environment' is the static scope in terms of the stack frames pointed to by the display registers, the 'control' is the code and the 'dump' is the current stack up to and including the previous dynamically linked stack frame.

Having illustrated these similarities it is interesting to compare AEs with their equivalent PASCAL code segments.

```
1.     Function f (x : real) : real;
          begin
             f := x * x;
          end;

       f (3) = {λx.x * x} (3)
           = 9

2.     Function g (Function h : real;  x : integer) : real;
          begin
             g := h(x) + h(x);
          end;

       g (sin, 3) = {λh.λx.h(x) + h(x)} (sin, 3)

                  = {λx.sin(x) + sin(x)} (3)
                  = sin(3) + sin(3)
                  = 0.28224

3.     Function g (y : real) : real;
          Function f (x : real) : real;
             begin
                f := x * x;
             end;
          begin
             g := f(y) + f(y);
          end;
       g (2) = {λy.{λf.f(y) + f(y)} (λx.x * x)} (2)
             = {λy.{λx.x * x} (y) + {λx.x * x} (y)} (2)
             = {λx.x * x} (2) + {λx.x * x} (2)
             = 4 + 4
             = 8
```

The preceding examples suggest that the constructs of a language such as PASCAL could be modelled in terms of AEs. These AEs could then be evaluated on the SECD machine to determine their outcome and thus the meaning or semantics of the PASCAL constructs. This approach to the formal semantics of a programming language is called operational in terms of the interpretation of the language's constructs by an abstract machine.

This method provides a further tool in that the interpreter itself could easily be altered in order to study the effect this will have on the semantics of the language in question. The most obvious example of such an application would be an investigation of the different mechanisms for passing arguments to functions i.e. by value, reference or name.

THE SECD MACHINE MECHANISM

In order to evaluate an AE using the SECD machine it must first be initialised in terms of a null evaluation stack S (hereafter referred to as the stack), an environment e.g. E = [x.2, y.4, '5'.5] i.e. the names x, y and '5' bound to the integers 2, 4 and 5 respectively, an AE e.g. C = [{λz.x + y + z} (5)] and a null dump D.

Brady (1977) proposes that the initialisation process be carried out by a function loadAE which loads an AE, C, and its corresponding environment E into an SECD machine state. The following function could be used :

```
loadAE(C, E) = iter (transform, <[], E, C, []>)
```

where iter is a recursive function that calls transform, the SECD machine interpreter. These functions will be described shortly.

Once the SECD machine has been initialised iter recursively calls transform which interprets the control and so doing advances the SECD machine to its next state. In this manner the machine proceeds through a number of SECD states driven by the control until the latter and the dump are both null. The head of the stack should then contain the value of the AE.

If the control is null and the dump is not null then a nested AE has just been evaluated and the dump must thus be popped. This is achieved by replacing the stack with the stack value that is on the dump with the head of the current stack pushed onto it. The latter corresponds to a value returned as a result of the evaluation of the nested AE. The tail of the current stack if it exists is lost. E and C are replaced by their respective values from the state which is on the dump. The current dump, D is replaced by the previous dump. In effect the previous dump is popped. The new control list is then interpreted.

An examination of the function transform will reveal that it only advances the SECD machine from the current state to the next. In order to continue the transformation process the function iter is recursive. The SECD machine state transitions described above are implemented by iter as shown below.

The notation employed to code these algorithms is based on McCarthy's formalism as described by Brady (1977). A set F of base or primitive functions, and a set C of functionals for building new functions out of old is assumed. The closure C(F) should then consist of all the computable functions in terms of the choice of F and C.

The set F can be devised in accordance with the nature of the set of problems which has to be solved i.e. numerical, textual, etc. The set C however is more specific and the set proposed by McCarthy is used viz. generalised composition of functions, conditional expressions and recursion. We note that the PASCAL conditional expression

```
If p1 then e1
else if p2 then e2;
```

is given by McCarthy as

```
(p1 → e1, p2 → e2, ..., pn → en)
```

where pn is often left out if pn is true.

```
iter (trans, state) =
    (null (C) →
        (null (D) → ( ), /* result on the stack evaluation complete */
            (S = h(S) :: S',   /* pop the previous state from the current D */
            E = E'
            C = C'
            D = D'
```

```
          iter (trans, trans (state))
         )
      ),
    iter (trans, trans (state))
   )
```

The interpretation of the control as performed by transform entails determining the value of the head of the control. If the head of the control list is :

1. an identifier then the value of the identifier is found by searching the current environment using the function lookup. The value found is pushed onto the top of the current stack and the control is replaced by its tail.
2. a λexp then a closure is contructed in terms of the current environment and the λexp and this value is pushed onto the stack. The control is replaced by its tail.
3. a combination i.e. rator/rand pair then the combination is popped off the control and the ap, the rator and the rand are pushed back onto the control in that order.
4. ap, a special object distinct from all AEs, then if :
   - **a.** the value which is at the top of the stack is a basic function (i.e. one which can be executed atomically) it is applied to the 2nd element of the stack. The result replaces the topmost two elements of the stack.
   - **b.** the value which is at the top of the stack is a closure then a new environment is created by prefixing the clos-env with the actual argument (2nd S) bound to the formal argument (clos-bv). The control is set to the value of the clos-body. The stack is cleared while the dump takes on the value of the SECD state just used to create the new one.

The underlying interpretation of the control is carried out by a function transform devised by Landin (1964). The version of transform given below is largely due to Brady (1977).

```
transform (<S, E, C, D>) =
   (let x = h (C)
   is-ident (x) → <lookup (E, x) :: S, E, t (C), D>,
   is-comb (x) → <S, E, C', D>
      where C' = sel-opd(x)::(sel-opr(x)::(ap::t (C))),
   is-lexp (x) → <make-clos (x, E) :: S, E, t (C), D>,

   x = ap →

      (let f = h (S)
      not is-clos (f) → < S', E, t (C), D>
         where S' = f (2nd (S)) :: t (t (S)),

      (let cl-body = sel-body (f)
       let cl-bv     = sel-bv (f)
       let cl-env    = sel-env (f)
       <[], extend (cl-env, 2nd (S), cl-bv), cl-body, D'>
         where D' = < t (t (S)), E, t (C), D>
      )))
```

A SIMPLE EXAMPLE

Consider the AE {λz.x + y + z} (5) in the environment [x.2, y.4, '5'.5] i.e. the identifiers x, y and '5' bound to the integers 2, 4 and 5 respectively.

For the purposes of evaluation we convert the infix expression to its prefix equivalent. We also introduce '+' as a one argument free variable which when applied to an integer i yields the function (+i). When (+i) is applied to an integer j, the integer, (i+j) is obtained. These transformations are described by Wegner (1968).

The call loadAE ({λz.(+ ((+x) y)) z} (5), [x.2, y.4, '5', 5]) results in the following SECD machine state trace:

```
<S, E, C, D>
<[], [x.2, y.4, '5', 5], [({λz.(+((+x) y)) z} (5))], []>
```

```
/* hC is a combination i.e. rator/rand pair */
<[],[x.2, y.4, '5', 5],[(5), {λz.(+((+x) y)) z}, ap], []>
/* hC is an identifier */
<[5], [x.2, y.4, '5', 5], [{λz.(+ ((+x) y)) z}, ap], []>
/* hC is a λ-expression, so make closA */
<[closA, 5], [x.2, y.4, '5', 5], [ap], []>
/* closA contains {λz.(+ ((+x) y)) z} and E, hC is ap so prepare to evaluate the
   closure by extending E with the cl-bv z bound to arg 5, set C = cl-body and push
   rest of the SECD state onto D = <[], [x.2, y.4, '5', 5], [], []> */
<[], [z.5, x.2, y.4, '5', 5], [(+ ((+x) y)) z], D>
/* hC is a combination i.e. (+ ((+x)y)) applied to z */
<[], E, [z, (+ ((+x) y)), ap], D>
<[5], E, [+ ((+x) y), ap], D>
<[5], E, [((+x) y), +, ap, ap], D>
<[5], E, [y, (+x), ap, +, ap, ap], D>
<[4, 5], E, [+x, ap, +, ap, ap], D>
<[2, 4, 5], E, [+, ap, ap, +, ap, ap], D>
/* hC is an identifier which is basic function */
<[+, 2, 4, 5], E, [ap, ap, +, ap, ap], D>
/* apply 1st S to 2nd S */
<[+2, 4, 5], E, [ap, +, ap, ap], D>
/* apply 1st S to 2nd S */
<[6, 5], E, [+, ap, ap], D>
/* the next three steps as before */
<[+, 6, 5], E, [ap, ap], D>
<[+6, 5], E, [ap], D>
<[11], E, [], [<[], [x.2, y.4, '5', 5], [], []>]>
/* C is null so pop D */
<[11], [x.2, y.4, '5', 5], [], []>
/* C and D both null, finish result on S */
```

CONCLUSION

As claimed by the title this work is intended as an introduction to the SECD machine and its use to evaluate applicative expressions. Hopefully the discussion and accompanying example will in some way fill in some of the gaps which I experienced when first coming to grips with the available literature. The work is also intended to serve as a basis for further study allied to AE's and the SECD machine.

ACKNOWLEDGEMENTS

I would like to thank Prof. S.W. Postma for suggesting the topic and for time spent reviewing the work.

REFERENCES

1. Landin, P.J. : The mechanical evaluation of expressions. Computer Journal, 6, 1964, pp 308-320.
2. Landin, P.J. : A λ-calculus approach in : Advances in Programming and Non-Numerical Computation. Fox, L. (Ed.), Pergamon, 1966.
3. Brady, J.M. : The Theory of Computer Science. A Programming Approach. Chapman and Hall, London, 1977.
4. Wegner, P. : Programming Languages Information Structures and Machine Organisation. McGraw-Hill, New York, 1968.

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 **11*** (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     This is the same length
         as the comment {This        as the comment rest assured.
         assured comment is the
         same length as the rest}    Comment is 50% of the text.
         rest assured.
```

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          It is always hard to type
         to a standard.Some people     to a standard.  Some people
         use different ones: single    use different ones:  single
         or     double   or no spaces  or double or no spaces
         before a sentence.  i think   before a sentence.  I think
         double looks best.            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 ~ove
- 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       Atinlay isay aay anguagelay hattay
       no gentleman should know,      onay entlemangay houlsay nowkay,
       but that every gentleman       utbay hattay everyay entlemangay
       should have forgotten.         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 87,661 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.

An Alternative Development of the Vienna Data Structures

N.C. PHILLIPS
University of Natal (Pietermaritzburg)
Department of Computer Science, Pietermaritzburg
3200

SYNOPSIS

This paper presents an axiomatic specification of a data type which is shown to be equivalent to the Vienna data structures. The advantage of this alternative approach is its obvious simplicity.

1. INTRODUCTION

The Vienna data structures, or Vienna objects, have been axiomized in [1] and elsewhere. In [1] the axioms are shown to be consistent by constructing a model for them, and the objects are then represented by rooted, edge-labelled trees with named terminal nodes. We can think of this work as an axiomatic specification of a data type which we shall call V-tree-μ. V-tree-μ consists of the Vienna objects with operators the selection functions and a path-constructing operator called μ in [1].

The intuition that the objects of this data type are determined by their paths may have suggested this specification in terms of of the operator μ. The intuition that they are determined by objects "one level down from the root" and by pointers to these sub-objects leads to the alternative approach taken here. We shall give an axiomatic specification of a data type which we shall call V-tree-λ. The principle difference between V-tree-λ and V-tree-μ is that V-tree-λ has a sub-object-constructing operator λ instead of μ. We claim the following advantages for the axiomisation of V-tree-λ.

(1) It is simple.
(2) It is easy to see how to construct a model for it.
(3) It is easily shown that the axioms are categorical in that given the set of atoms, they determine the data type up to isomorphism.

Of course, V-tree-μ and V-tree-λ are different as data types since they have different operators, but with a little effort we shall show that they are equivalent in the sense that λ is definable in V-tree-μ, that μ is definable in V-tree-λ and, from a given set of atoms, V-tree-μ and V-tree-λ define the same set of objects.

2. SPECIFICATION OF V-TREE-μ AND V-TREE-λ

Each data type is specified in terms of two non-empty sets - a set S whose elements are called selectors and a set O whose elements are called objects. Furthermore, for each $s \in S$ there is a corresponding function $\bar{s}$ from O to O called a selection function. If $A \in O$ we shall write sA for $\bar{s}(A)$.

For V-tree-μ only, we construct (S^*, I), the free monoid generated by S with identity element I and with the monoid operation denoted by concatenation. Elements of S^* are called composite selectors. In addition, for each $\kappa \in S^*$ there is a corresponding selection function $\bar{\kappa}$ defined on O. If $A \in O$ we shall write κA for $\bar{\kappa}(A)$. (In [1] no distinction is made between κ and $\bar{\kappa}'$.)

In what follows: $s, s_1, s_2, \ldots$ denote selectors, $\kappa, \kappa_1, \kappa_2, \ldots$ denote composite selectors and A, B, C, ... denote objects.

Axioms, lemmas and definitions prefixed by μ are for the data type V-tree-μ and those prefixed by λ are for the data type V-tree-λ. Apart from minor rewriting we follow [1] closely for V-tree-μ. Theorems and lemmas about V-tree-λ which are proved in [1] will not be proved again here.

μ **Axiom 1.** $sA \in O$

μ **Axiom 2.** $(\kappa s)A = \kappa(sA)$

μ **Axiom 3.** $IA = A$

μ **Axiom 4.** $(\exists A)(\forall s)[sA = A]$

μ **Axiom 5.** $(\forall s)\ (sA = A) \Rightarrow (\forall B)(\exists\kappa)(\kappa B = A)$

μ **Lemma 1.** $(\exists 1A)(\forall s\ )[sA = A]$

λ **Axiom 1.** $(\exists 1A)(\forall s\ )[sA = A]$

The object satisfying μ Lemma 1 and λ Axiom 1 is called the empty object and is denoted by Ω.

For both V-tree-μ and V-tree-λ we define the set of atoms, A , to be $\{A : (\forall s)(sA = \Omega)\}$ and the set of elementary objects, ξ, to be $A\ - \{\Omega\}$. We assume that ξ is not empty and that no ordered pair is an element of an atom. In what follows, e, $e_1,\ e_2,$ ... will denote elementary objects.

μ **Axiom 6.** $(\forall\kappa)(\forall e)[\kappa A = e \Leftrightarrow \kappa B = e] \Rightarrow A = B$

λ **Axiom 2.** If $A \notin A$ then $(\forall s)(sA = sB) \Rightarrow A = B$

For V-tree-μ we define: κ_1, κ_2 are dependent iff there is a τ in S^* such that $\kappa_1 = \tau\kappa_2$ or $\kappa_2 = \tau\kappa_1$.

If $\kappa_1,\ \kappa_2$ are dependent we write dep(κ_1, κ_2).

μ **Axiom 7.** $(\exists B)[\kappa B = e \cap (\forall\tau)(\sim\text{dep}(\kappa,\tau) \Rightarrow \tau B = \tau A)]$

λ **Axiom 3.** If $sA = \Omega$ and $C \neq \Omega$ then $(\exists B)\ (\forall s_1)[s_1B =$ if $s_1 = s$ then C else $s_1A]$

μ **Lemma 2.** The B satisfying μ axiom 7 is unique.

λ **Lemma 3.** The B satisfying λ axiom 3 is unique.

Proof.

Suppose that $sA = \Omega$ and $C \neq \Omega$ and that B_1 and B_2 satisfy λ axiom 3. Since $sB_1 = C$ and $C \neq \Omega$, $B_1 \notin A$, so $B_1 = B_2$ by λ axiom 2.

The B satisfying μ axiom 7 is denoted $\mu(A,\kappa,e)$.

The B satisfying λ axiom 3 is denoted $\lambda(A,s,C)$.

μ **Axiom 8.** O is the closure of A under μ.

λ **Axiom 4.** O is the closure of A under λ.

For ease of reference later, we now list the axioms for each data type.

V-tree-μ

μ1. $sA \in O$

μ2. $(\kappa s)A = \kappa(sA)$

μ3. $IA = A$

μ4. $(\exists A)(\forall s)[sA = A]$

μ5. $(\forall s)\ (sA = A) \Rightarrow (\forall B)(\exists \kappa)(\kappa B = A)$

μ6. $(\forall \kappa)(\forall e)[\kappa A = e \Leftrightarrow \kappa B = e] \Rightarrow A = B$

μ7. $(\exists B)[\kappa B = e \cap (\forall \tau)(\sim\mathrm{dep}(\kappa,\tau) \Rightarrow \tau B = \tau A)]$

μ8. $\mathcal{O}$ is the closure of $\mathcal{A}$ under μ.

V-tree-λ

λ1. $(\exists 1A)(\forall s\ )[sA = A]$

λ2. If $A \notin \mathcal{A}$ then $(\forall s)[sA = sB] \Rightarrow A = B$

λ3. If $sA = \Omega$ and $C \neq \Omega$ then $(\exists B)\ (\forall s_1)[s_1B = \text{if } s_1 = s \text{ then } C \text{ else } s_1A]$

λ4. $\mathcal{O}$ is the closure of $\mathcal{A}$ under λ.

3. PROPERTIES OF V-TREE-λ

λ Definition 1:

$A^* = \text{if } A \in \mathcal{A} \text{ then } A \text{ else } \{(s,C) : sA = C \cap C \neq \Omega\}$
A* will be called the character of A.

λ Lemma 4.

(i) $A^* = B^* \Rightarrow A = B$.

(ii) $(sA)^* = \text{if } (\exists C)(s,C) \in A^* \text{ then } C \text{ else } \Omega$.

(iii) If $sA = \Omega \cap C \neq \Omega$ then $\lambda(A,s,C)^* = \text{if } A \in \mathcal{A} \text{ then } \{(s,C)\} \text{ else } A^* \cup \{(s,C)\}$.

Proof. Recall our assumption that no ordered pair is an element of an atom.

(i) We show first that if one of A,B is an atom and the other is not then $A^* \neq B^*$. For, if B is not an atom there is an s and a $C \neq \Omega$ such that $sB = C$. Then $(s,C) \in B^*$. But if A is an atom then $A^* = A$, so $A^* \neq B^*$ since $(s,C) \notin A$. Now suppose that $A^* = B^*$. If A, B are atoms, A = B follows by definition of character; if A,B are not atoms, A = B follows from λ axiom 2.

(ii) This follows at once from the definition of character (consider two cases, $A \in \mathcal{A}$ and $A \notin \mathcal{A}$).

(iii) This to follows easily from the definition of character and the fact that $s_1\lambda(A,s,C) = \text{if } s_1 = s \text{ then } C \text{ else } s_1A$.

The above lemma suggests that for a model of V-tree-λ we construct all possible characters. λ Axioms 3 and 4 suggest that characters are atoms or are non-empty finite sets $\{(s_1,C_1),\ldots,(s_n,C_n)\}$ where $s_1,\ldots,s_n$ are distinct selectors and $C_1,\ldots,C_n$ are non-empty objects. Accordingly we recursively define the set $\mathcal{O}$ of objects to be the smallest set which contains $\mathcal{A}$ and all finite sets described above. Next we need to define the selection functions and the function λ on $\mathcal{O}$. Again the above lemma suggests how this should be done.

We define: $sA = \text{if } (\exists C)(s,C) \in A \text{ then } C \text{ else } \Omega$, and if $sA = \Omega$ and $C \neq \Omega$ we define: $\lambda(A,s,C) = \text{if } A \in \mathcal{A} \text{ then } \{(s,C)\} \text{ else } A \cup \{(s,C)\}$.

It is now easy to check that $\mathcal{O}$, with selection functions and the operator λ defined above satisfies λ axioms 1 - 4.

Theorem 1. Any two models of V-tree-λ with the same set of atoms are isomorphic.
Proof.

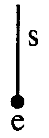
λ Lemma 4 and λ axiom 4 show that any model of the λ axioms will be isomorphic to the model of characters under the map which takes each object to its corresponding character.

In [1] some pains are taken to represent objects as tress. We can see no harm in directly defining a V,DL tree to be the character of an object. We give below some pictures of VDL trees.

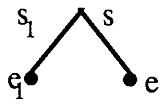
Object Tree

e

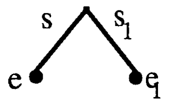
$A = \lambda(\Omega,s,e)$



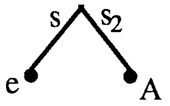
$B = \lambda\ (A,s_1,e_1)$



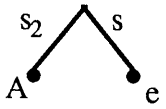
or



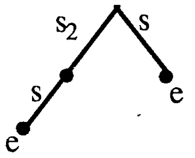
$C = \lambda\,(A,s_2,A)$



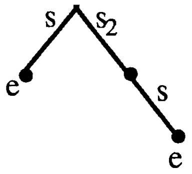
or



or



or



Ω is the empty tree, its tree is the empty picture.

4. THE EQUIVALENCE OF V-TREE-μ AND V-TREE-λ

We need a few more facts about V-tree-μ before we can proceed. In V-tree-μ the character set, $\overline{A}$, of A is defined to be $\{(\kappa,e) : \kappa A = e\}$.

μ Lemma 5.

(i) $\overline{A} = \overline{B} \Rightarrow A = B$.

(ii) if $\kappa_1\ A = e = \kappa_2 A$ and $\kappa_1 \neq \kappa_2$ then $\sim\text{dep}(\kappa_1,\kappa_2)$

(iii) $\overline{\mu(A,\kappa,e)} = \{(\kappa,e)\} \cup \{(\tau,e) : \tau(A) = e \cap \sim\text{dep}(\tau,\kappa)\}$.

(iv) $\overline{sA} = \{(\tau,e) : (\exists\kappa)[(\kappa,e) \in \overline{A} \cap \kappa = \tau s]\}$.

Proof.

(i)-(iii) are proved in [1]. For (iv) observe that $(\tau,e) \in$ sA iff $\tau(sA) = e$ iff $\kappa A = e$ where $\kappa = \tau s$.

Theorem 2. λ is definable in V-tree-μ.

Proof:

Let $sA = \Omega$ and $C \neq \Omega$. We need to show in V-tree-μ that there is a unique object which satisfies λ axiom 3. By μ axiom 8, the characteristic set $\overline{C}$ of C is finite, and it is non-empty because $C \neq \Omega$.

Let $\overline{C} = \{(\kappa_1,e_1),\ldots,(\kappa_n,e_n)\}$.

Define: $B_0 = A$ and $B_m = \mu(B_{m-1},\kappa_m s,e_m)$ for $1 \leq m \leq n$.

Note that, since $sA = \Omega$, if $t(A) = e$ then $\sim\mathrm{dep}(t,\kappa_m s)$ for $1 \leq m \leq n$. In particular when $m = 1$ this fact and μ Lemma 5 (iii) show that $\overline{B_1} = \{(\kappa_1 s,e_1)\} \cup \overline{A}$.

Assume that $1 \leq m \leq n$ and $\overline{B_m} = \{(\kappa_1 s,e_1),\ldots,(\kappa_m s,e_m)\} \cup \overline{A}$.

Note that by μ Lemma 5 (ii), $\sim\mathrm{dep}(\kappa_j s)$ for $1 \leq j \leq m$.

By μ Lemma 5 (iii) again,

$\overline{B_{m+1}} = \{(\kappa_{m+1}s,e_{m+1})\} \cup \{(\tau,e) : \tau(B_m) = e \cap \sim\mathrm{dep}(\tau,\kappa_m s)\}$

$= \{(\kappa_{m+1}s,e_{m+1})\} \cup \overline{B_m}$ by the induction hypothesis and the noted facts about dependence

$= \{(\kappa_1 s,e_1),\ldots,(\kappa_{m+1}s,e_{m+1})\} \cup \overline{A}$ by the induction hypothesis again.

Write B for B_n. We have shown that $\overline{B} = \{(\kappa_1 s,e),\ldots,(\kappa_n s,e_n)\} \cup \overline{A}$.

Hence, since $sA = \Omega$ and by μ Lemma 5 (iv), $\overline{sB} = \{(\kappa_1,e_1),\ldots,(\kappa_n,e_n)\} = \overline{C}$.

So $sB = C$ by μ Lemma 5(i).

From the above expression for B and μ Lemma 5 (iv) we see that if $s_1 \neq s$ then $\overline{s_1 B} = \overline{s_1 A}$, so $s_1 B = s_1 A$.

Thus B satisfies λ axiom 3.

Finally we must prove the uniqueness of B. Suppose that D satisfies λ axiom 3.

If $\kappa = \tau s$ then $\kappa D = e$ iff $\kappa B = e$ because $\overline{sD} = \overline{sB} = \overline{C}$.

If $\kappa = \tau s_1$ and $s_1 \neq s$ then $\kappa D = e$ iff $\kappa B = e$ because $\overline{s_1 D} = \overline{s_1 B} = \overline{A}$.

Therefore, by μ axiom 5 and since D and B are not atoms, $D = B$.

Theorem 3. μ is definable in V-tree-λ.

Proof. First we extend V-tree-λ by defining a function $\overline{\kappa}$ for each composite selector κ in accord with μ axioms 1-3. To do this we simply take μ axiom 3 as a definition and μ axiom 2 as a recursive definition of $\overline{\kappa s}$.

We now prove in V-tree-λ that given A,κ,e there is a unique B which satisfies μ axiom 7.

If $\kappa = I$ then taking e for B satisfies μ axiom 7.

Suppose that $\kappa = s_1,\ldots,s_n$.

Define $B_0 = e$ and $B_m = \lambda(\Omega,s_m,B_{m-1})$ for $1 \leq m \leq n$.

If $A^* = A$ then $B = B_n$ satisfies μ axiom 7.

Suppose that $A^* = \{(s^1{}_1,C_1),\ldots,(s^1{}_m,C_m)\}$.

If $s_n \neq s^1{}_j$ for $1 \leq j \leq m$ then $B = \lambda(A,s_n,B_{n-1})$ satisfies μ axiom 7.

Suppose $s_n = s^1{}_j$: we may assume without loss of generality that $j = 1$.

If $m = 1$ then again $B = B_n$ satisfies μ axiom 7.

If $m > 1$ then define $D_2 = \lambda(\Omega,s^1{}_2,C_2)$ and $D_{i+1} = \lambda(D_i,s^1{}_{i+1},C_{i+1})$ for $1<i<m$.

Now $B = \lambda(D_m,s_n,B_{n-1})$ satisfies μ axiom 7.

It remains to prove uniqueness. Suppose D satisfies μ axiom 7.

Consider any τ such that $\tau \neq \kappa$ and $\tau D \in \xi$.

$\tau \neq \tau_1 k$ since otherwise $\tau D = \tau_1 \kappa D = \tau_1 e = \Omega$

$k \neq \tau_1 \tau$ since otherwise $kD = \tau_1 \tau D = \Omega$ since $\tau D \in \xi$.

Thus $\sim\mathrm{dep}(\tau,\kappa)$, so $\tau D = \tau A = \tau B$.

We have shown: $(\forall\tau)(\forall e)[\tau D = e \leftrightarrow \tau B = e] \Rightarrow A = B$.

Now $D = B$ follows from λ Lemma 6 which we prove below.

λ Lemma 6. $(\forall\tau)(\forall e)[\tau B = e \leftrightarrow \tau B = e] \Rightarrow A = B$

Proof. λ Axiom 4 justifies the following recursive definition of depth:

depth A = if $A \in \mathcal{A}$ then 0 else $1 + \max\{\text{depth } sA : sA \neq \Omega\}$.

Suppose $(\forall\tau)(\forall e)\ [\tau A = e \leftrightarrow \tau B = e]$.

If $A = \Omega$ then $A = B$ since $A = \Omega \Leftrightarrow (\forall\tau)\ [\tau A = \Omega]$.

If $A \in \xi$ then $A = B$ since $(\forall\tau)[\tau A \neq \Omega \Leftrightarrow \tau = I]$.

Thus $A = B$ if depth $A = 0$. Suppose depth $A > 0$.

For any s, $(\forall\tau)(\forall e)[\tau sA = e \Leftrightarrow \tau sB = e]$ so by our induction hypothesis, $(\forall s)[sA = sB]$. So by λ axiom 2, $A = B$.

Theorem 4. Given $\mathcal{A}$, V-tree-μ and V-tree-λ determine the same set of objects (up to isomorphism).

Proof. In view of theorems 2 and 3 it remains to show that the closure of $\mathcal{A}$ under μ is the closure of A under λ. But theorem 2 implies that the closure of $\mathcal{A}$ under λ is contained in the closure of $\mathcal{A}$ under μ and theorem 3 implies that the closure of $\mathcal{A}$ under μ is contained in the closure of $\mathcal{A}$ under λ.

5. CONCLUSION

There are close connections between the work presented here and other descriptions of the Vienna objects [2, 3]. The axioms for V-tree-λ are similar to the "ground axioms" in Standish [4]. Models of V-tree-λ are precisely the "constructive models" of [4]. Section 4 above gives the precise relationship between the Vienna objects and the "constructive models" of [4].

The VHL language QUADLISP [6] which is under development at the University of South Africa requires tree-like objects which are considerably more complicated than the VDL trees. The selector which pointed me to this work was invoked when S.W. Postma asked me for a specification of the QUADLISP objects.

REFERENCES

1. OLLONGREN, O. (1974). Definition of Programming Languages by Interpreting Automata,

A.P.I.C. Studues in Data Processing No 11, Academic Press, 87-116.

2. LUCAS, P., LAUER, P. and STIGLEITNER, H. (1968). Method and Notation for the formal Definition of Programming Languages, TR 25.087, IBM Research Laboratory, Vienna.
3. WEGNER, P. (1972). The Vienna Definition Language, Computing Surveys, 4, No 1, 5-63.
4. STANDISH, T.A. (1978). Data Structures - an Axiomatic Approach, Current Trends in Programming Methodology, Vol IV Data Structuring, Prentice-Hall, 30-59.
5. POSTMA, S.W. (1980). Lyspe/2 a Programming Language for Theoretical Computer Science, pp 39-75 in Matuszewski, R. Conference on Technology of Sciernce, Plock, Poland, 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:
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 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. 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.