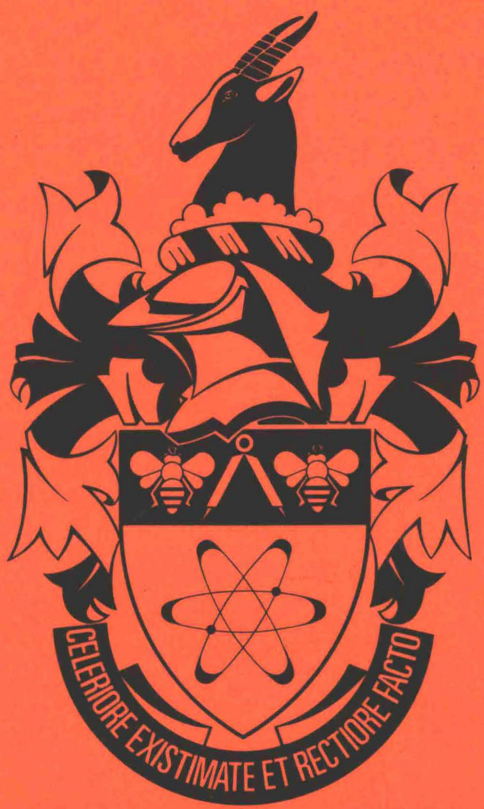


Quaestiones Informaticae

Vol. 2 No. 2 May, 1983

Quaestiones Informaticae

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

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.

NOTE FROM THE EDITOR

Three points must be made by way of introduction to the second issue of Volume 2 of Quaestiones Informaticae.

Firstly, an apology is in order for the mistake in the date (November 1983 instead of 1982) at the foot of my note introducing the preceding issue. Lacking the services of a professional proof reader, printing errors are bound to show up from time to time, but it is hoped that their number will be kept to a minimum!

Secondly, it is a pleasure to announce that this journal will not only serve to publish papers of a scientific or technical nature on computing matters under the auspices of the Computer Society of South Africa. An agreement has been reached to share the facilities of Quaestiones Informaticae between the CSSA and SAICS, the South African Institute of Computer Scientists. Henceforth this journal will also be used to publish the Transactions of this Institute. This implies certain changes to the cover pages which will be implemented in future issues. I shall continue to serve as editor, but on behalf of SAICS Prof R. J. van den Heever will share some of my duties and act as co-editor.

Finally Mr Edwin Anderssen, of Rand Afrikaanse Universiteit, has agreed to serve as circulation manager for Quaestiones Informaticae. I am grateful indeed that he is willing to serve the journal in this capacity, and look forward to a long period of fruitful cooperation.

G WIECHERS

May, 1983

Case-Grammar Representation of Programming Languages

Judy Mallino Popelas and Peter Calingaert
University of North Carolina at Chapel Hill, USA

Abstract
The correction of errors in programs can be based on an analysis, that subordinates syntactic relationships to functional relationships among elements of a program. For this purpose, *case grammars,* originally developed to model natural languages, have been adapted to model programming languages. The component parts of such a modified case grammar are described, and a case grammar for a subset of Pascal presented.

INTRODUCTION

The laudable goal of ensuring that a program is correct before it is presented to a computer is an elusive one even for experienced professional programmers. It is virtually unattainable for the rapidly increasing masses of persons for whom some computer programming is a necessary, but nevertheless part-time, activity. Errors occur both in *programming,* the design of an algorithm and the selection of data structures, and in *coding,* the representation of the algorithm and data structures in a programming language (PL). Although programming errors are of great importance, our research has focussed on the more mundane, but very aggravating, errors in coding.

Humans surely spend much less time encoding computer programs than they do communicating with one another by means of natural language (NL). We hypothesize that many of the errors humans make in encoding programs are similar to those they make in encoding thoughts into NL utterances. This suggests that techniques for correcting encoding errors in NL should help to correct coding errors in PL, and that useful models of NL representation should lead to useful models of PL representation.

Communication between humans can proceed effectively even when the utterances violate rules of syntax. Thus, "you was coming" and "they gave it to John and I" are clearly understandable, although incorrect. Even "today me shirt buy" is far from incomprehensible. The human who hears such an utterance does not immediately reject it because of its faulty syntax. He tries instead to understand it, using whatever nonsyntactic clues he can find. In translating computer programs with syntax errors, the compiler, too, can be made to use nonsyntactic clues to determine the underlying meaning when normal syntax correction would fail.

A particularly attractive model of NL, well capable of representing the meaning of the syntactically incorrect utterances presented in the previous paragraph, is the *case grammar* of Fillmore[1]. Case grammar (CG) concentrates on the underlying deep structure by associating with each verb a *case frame.* The case frame is occupied by one or more phrases, each of which plays a specific role demanded by the associated verb.

Thus the verb "buy" requires an agent who buys (a phrase in the *agentive* case) and an object that is bought (a phrase in the *objective* case). The phrases must often possess specified attributes; the agent of "buy" must be animate. In the example "today me shirt buy" the case frame requirement for an agent is filled by "me" and the case frame requirement for an object by "shirt".

Meaning can be extracted from the NL utterance without performing conventional syntactic analysis, by identifying the phrases that occupy the case frames. By adapting CG to PLs, which are much less complex than NLs, we expect to improve our ability to correct coding errors, and to perform correction without conventional syntactic analysis.

We have begun by developing a CG for Pascal and an algorithm for translating a syntactically incorrect program into its CG representation. We chose Pascal because it is formally defined [2], designed for efficiency of conventional translation [3], and widely used. Our adaptation of CG to PL is presented in Section 2, and illustrated in Section 3 by a subset of the Pascal grammar.

CASE GRAMMARS FOR PROGRAMMING LANGUAGES

Overview

Like their NL counterparts, CGs for programming languages emphasize functional rather than positional relationships between object phrases and verbs. Again like Fillmore's CGs, they emphasize an object's attributes rather than its form.

CGs for programming languages, hereafter referred to simply as CGs, have three basic components. The first component, an object space denotation, defines the objects found in a language in terms of attributes and attribute combinations. The second component, the verb dictionary, contains a *case frame entry* for each verb in the language. Each case frame entry is composed of a *header* and a *case frame.* The header gives the attributes of the object resulting from the filled case frame. The case frame itself describes the objects required by the verb, together with the functional relationship, or *case relation,* between each object and the verb. The constituent object descriptions in a case frame are called *case frame slots,* or more generally, *frame slots.* The final CG component, an access form dictionary, is similar to the verb dictionary. It is composed of *form frame entries,* which consist of a header and a *form frame.* Each form frame defines the access to an object other than a simple literal, by describing the objects that compose it or can be converted into it. Again, the object descriptions in a form frame are called frame slots, or more specifically, *form frame slots. Form relations* between the component objects and the resultant object may be specified. However, unlike case relations, which are often indicated explicitly via keywords and other PL markers, form relations generally are not indicated explicitly in PLs. Any keyword or PL marker that indicates a case or form relationship is called a *case* or *form relation predictor.* The header and frame of a case or form frame entry are analogous to the left- and right-hand sides of a context-free production.

Object Space Denotation

We use the word "object" to denote any entity that can be referred to or manipulated within the context of a given PL. Integer and real numbers, for example, are objects in most PLs. An object space is defined in large part by the attributes pro-

vided in a language, and by the combinations of attribute values that the objects in the language can possess. Attributes can be classified as being *universal* if they apply to every PL object, or *dependent* if they apply to only a subset of the PL objects.

Usage, class, and structure are three universal attributes. Access is a fourth universal attribute, but it is an attribute more of the frame slots than of the objects that satisfy frame slots. Usage refers to the way in which an object can be used. 'Value' usage indicates that an object can be used only as a value, whereas 'variable' usage indicates that an object can be used both as a value and as a store.

To be used, objects must be accessible. Commonly provided access methods include naming, referencing, direct representation, generation, and modification. Naming is one of the most common. Names, which have no inherent meaning, must be bound to a particular object before they can be used to refer to it. Languages commonly provide declaration parts or declaration statements for this purpose. References can be regarded as machine-generated names. They are usually used to refer to dynamically created variables. Direct representation differs from naming in that it is a permanent association between representation and object. In most languages, for example, '5' always represents the integer 5. Generation involves the execution of a sequence of one or more operators. The expression '2+4' generates the integer 6. Access by modification refers to the methods commonly used to refer to objects such as array or record components. Modification is similar to generation, except that no explicit operator is present.

Some access-usage combinations can be referred to by a single word. 'Literal', for example, refers to objects accessed by direct representation and used as values. Conversely, other access-usage combinations may encompass several distinguishable kinds of objects. Both array components and record components, for example, are accessed by modification and used as variables.

A *class* is defined as a set of scalar values and a set of case and form frame slots that accept those values. The restriction to scalar values effectively separates the concepts of class and structure, which together provide a complete and minimal set of concepts for describing any type of object. Some common classes of objects are integer, real character, boolean, verb, [verb], procedure, function, name, pointer, label, and class attribute. The notation '[verb]' stands for a verb together with the objects it requires. This constitutes a completed case frame (*i.e.*. a programming language statement or expression).

Most languages allow for structured as well as scalar objects. A structured object does not have a single associated class attribute. Rather, each of its component objects, if scalar, has an associated class attribute.

Some objects may have dependent attributes in addition to the four universal attributes. Exactly which dependent attributes an object possesses is determined by the value of some other attribute. Only objects whose class attribute is 'real', for example, have a precision attribute.

Specification of Object and Object-Phrase Requirements

An object phrase consists of one or more objects, plus preceding keywords and surrounding punctuation. Both case and form frame slots specify object-phrase requirements. Objects are the most important components of object phrases. Their requirements may be specified by stating permissible structure, class, usage, and access attribute values, as well as dependent attribute values. For example, the specificatin 'scalar, real,value,direct_representation' will be satisfied by objects like 1.0, 5.37, *etc.* The combination access-usage specification 'literal' can replace the separated specification 'value,direct_representation'. Some attributes may remain unrestricted. The specification 'scalar,real,value' places no restriction on the access method. Note that, since a variable can be used as a value, it satisfies a usage specification of 'value'. To force a restriction to non-variable objects, one could either specify 'value_only', or use an access-usage combination such as 'literal'. Alternative attribute values may be specified, as in 'scalar,integer|real,value'. Restrictions on the values of dependent attributes are specified in parentheses after the attribute value they modify, as in 'scalar,real(single_precision),value'.

Punctuation symbols are classified as predecessors if they precede objects, brackets if they bracket objects, separators if they separate like objects, and successors if they succeed objects. A *simple object phrase* is defined as a keyword, which acts as a case or form relation predictor, followed by a predecessor, a left bracket, an object or a sequence of like objects separated by separators, a right bracket, and a successor, in that order. Of these components, only a single object is mandatory. The text *'with* a, b, c' represents a simple object phrase. The keyword '*with*' is a case relation predictor; 'a', 'b', and 'c' represent like objects (record variables); and ',' acts as a separator.

To specify a simple object phrase, first specify the object, as already described. Attach a superscript to the object specification to indicate the number of like objects permitted in the object phrase. The superscript '+' indicates one or more like objects; '*' indicates zero or more like objects; 'op' indicates an optional object; and a positive integer indicates that number of like objects. The default value is unity. An encoding of the surrounding punctuation, enclosed by parentheses, is also attached as a superscript. A *case label*, denoting both the functional relationship (case relationship) of the object to the verb, and the particular keyword or other PL symbol, if any, that acts as the case relation predictor, is attached as a subscript. The case relation predictor appears parenthesized, after the case relationship. Some common case relationships are *indicant,* which specifies a name object used by a verb that binds names to other objects (variable declaration statements have indicant objects); *selector,* which indicates an object used by a verb to select among many possible objects (GOTO, IF, and CASE statements have selector objects); *donor,* which indicates an object whose value is given to another object (assignment statements have donor objects); and *objective,* a general case relationship that indicates an object that receives the action of a verb (operators such as +, −, and * have objective case objects). Thus, the specification

$$\text{record,variable}^{+\ (00,0)}_{\text{selector}(\textit{with})}$$

is satisfied by the simple object phrase 'with a, b, c', assuming 'a', 'b', and 'c' are the names of record variables. Note that '0' is used in the punctuation encoding to indicate the absence of a predecessor, brackets, and a successor. The generic form

$$\text{OBJECT}^{\text{MULT PUNCT}}_{\text{CL}}$$

where OBJECT represents an object specification, MULT a specification of the number of like objects in the object phrase, PUNCT the punctuation encoding, and CL the case label, describes a simple object phrase specification.

A *complex object phrase* is defined as a case or form relation predictor, followed by a predecessor, a left bracket, one or more object phrases (simple or complex), or a repeated sequence of one or more object phrases separated by separators, a right bracket, and a successor, in that order. Of these components, only a single object phrase is mandatory. The text '[1:10]' represents a complex object phrase, where '1:' and '10' represent simple object phrases, and '[' and ']' are used as brackets. To specify a complex object phrase, first parenthesize the interior object phrase specification(s). Then attach the subscripts and superscripts to the parenthesized specification(s), in the same way as for a simple object phrase. The complex object phrase specification

(scalar,integer,literal$^{op(000:)}$
scalar,integer,literal)$^{+\ (0|],0)}$

is satisfied by either of the complex object phrases '[1:10]' and '[5,2:10]', and by many others as well. The generic form

$$(OBJ1 \ldots OBJN)^{MULTI\ PUNCT}_{CL}$$

describes the specification of a complex object phrase. OBJ1 ... OBJN represent specifications of simple or complex object phrases, and MULT, PUNCT, and CL represent exactly what they do in the specification of a simple object phrase.

Verb Dictionary

The verb dictionary contains one case frame entry for each verb in the language. Case frames are enclosed by square brackets. They contain specifications for each object phrase required to the verb. They also indicate the case relation of each object phrase phrase to the verb, even if that relation is not made explicit by a keyword or other PL marker. Following is a case frame entry for the GOTO verb.

scalar,[verb](GOTO,active,imperative,regular),literal
[scalar,label,value$_{selector\ (goto)}$]

GOTO requires one object, a label, which acts as a selector. The keyword *'goto'* precedes or predicts the selector object. CASE and IF statements also require selector objects, although these are predicted by different keywords.

In most languages, verbs and their corresponding completed case frames ([verb]s) will have significant dependent attributes. Four such attributes are discussed here: name, voice, mode, and influence. Name simply identifies the verb, as shown for GOTO. Voice may be 'active' or 'passive'. Passive voice indicates a verb, like the variable-creation verb, that can be executed at most once. Active verbs may be executed repeatedly. Verb mode may be 'imperative' or 'operator'. Operators include verbs like addition, multiplication, and binary selection (*i.e.,* the IF statement verb) that result in a single object. Imperative verbs, like assignment, result in changes to the environment. Verb influence may be 'regular' or 'meta'. 'Meta' indicates that the verb requires objects that are themselves statements. The case frame entry for the metaverb BINARY_SELECTION, without interior labels, is the following.

scalar,[verb](BINARY_SELECTION,active,
operator(scalar,[verb](,active,imperative,)),meta), literal
[scalar,boolean,value$_{selector(if)}$
[verb](,active,imperative,)$_{objective(then)}$
[verb](,active,imperative,)$^{op}_{objective(else)}$]

Besides the selector object, BINARY_SELECTION requires either one or two statement objects that are in the objective case, which receives the action of the verb. The objects must be active, imperative statements. The blank in the name and influence attribute positions indicates that any value for those attributes is acceptable. IF and other meta operator statements also satisfy the requirements because they ultimately generate active, imperative statements. Note that BINARY_SELECTION, because it is an operator, has dependent attributes that specify the structure and class of its resultant, generated object.

Verbs that require multiple objects often require agreement among two or more of them. We introduce *attribute variables* to express interobject dependencies. The attribute variables used in a given frame are implicitly created at the beginning of the frame, and remain accessible throughout the frame. Attribute variables are implicitly assigned values when they prefix an attribute restriction in an object specification. The specification 'scalar,CLSI:integer|real,value' causes the value of the class attribute of the object satisfying the specification to be assigned to the attribute variable CLSI. Attribute variables can be used without a specific attribute restriction, as in 'scalar, CLSI: , value'. The blank following 'CLSI:' indicates that there is no restriction placed on the class attribute. The colon indicates that the variable CLSI is to be assigned a value. When attribute variables appear without a succeeding colon, they specify a restriction to whatever attribute value they currently possess. Consider the case frame entry for assignment.

scalar,[verb](ASSIGN,active,imperative,regular),literal
[STRI: ,CLSI: ,variable$_{recipient}$
STRI,CLSI,value$_{donor(:=)}$]

When the recipient object phrase is encountered, the attribute variables STRI and CLSI are assigned values. By using STRI and CLSI to specify attribute values for the donor object phrase, agreement between the two objects is forced.

Conditional clauses may be used to modify a succeeding attribute value specification, simple object-phrase specification, or complex object-phrase specification. They consist of a predicate enclosed by '(= =)' brackets. The case frame for an ASSIGN verb that allows integers to be assigned to reals can be specified by using a conditional clause.

scalar,[verb](ASSIGN,active,imperative,regular),literal
[STRI: ,CLSI: ,variable$_{recipient}$
STR,CLSI| (=CLSI=real=) integar, value$_{donor(:=)}$]

Access Form Dictionary

The access form dictionary defines the access methods. For example, a complex literal like a procedure would have an access frame describing each of its component object phrases. For array components, which are accessed by modification, the access frame describes the array object and the objects that could be used as indices. Access frames specify transformations of objects to other objects.

Access frames are enclosed in angular brackets. Following is a simplified form frame entry for name access.

STRUCT, CLASS, USAGE, named
<name(bound(STRUCT: ,CLASS: ,USAGE:))>

It states that a bound name object may be transformed into an object whose structure, class, and usage attribute values are determined by the dependent attributes of 'bound'.

Semantics

Because a case grammar deals with language at the object level rather than at the symbol level, at least a partial definition of semantics is needed to make it complete. The semantics must specify the creation of objects, the association of attributes with objects, and the deletion of objects. A complete notation for defining the semantics in a case grammar for Pascal is given in the first author's dissertation[4]. The details of the notation are unimportant, since many other notations would have served as well. However, a simplified subset is presented here to enable the reader to understand the case grammar example presented in the next Section.

Semantic action statements are used to assign values to attribute variables explicitly. They are of the form 'attribute variable <— value', and are enclosed by '{}' brackets. They may appear anywhere in a case or form frame.

SYMTAB is a global attribute variable, accessible from any frame. It contains the kind of information commonly found in symbol tables, the association of names with the objects they represent. In particular, the value of SYMTAB will be a sequence of name literals, each with its associated dependent binding attribute. The value 'bound' has, in turn, three associated dependent attributes: structure, class, and usage. These give the structure, class, and usage of the object to which the name has been bound. Both 'unused —> (unbound)' and 'used —> (bound(scalar,integer,variable))' represent legitimate entries in SYMTAB. The operator '+||' will be used to add entries to SYMTAB. Similarly, '−||' is used to delete entries from SYMTAB. The verbs CREATE_PROG and CREATE_VAR demonstrate the +|| operation.

EXERPTS FROM A CASE GRAMMAR

We present here excerpts from a case grammar for a very small subset of Pascal. The subset includes an abvreviated program statement, the *var, begin,* assignment, *if,* and *while*

statements, and several operators. Labels are omitted. The boolean entities 'true' and 'false' are treated as literal values the lexical structure of simple literals such as integers and reals.

Object Space Denotation

The object space is defined by three tables. Table 1 lists the attributes used in the grammar, together with the values they may assume, Table 2 lists attribute dependencies, and Table 3 shows the co-occurrence of attribute values in objects. Because the language has only scalar objects, neither Table 1 nor the rest of the grammar includes a structure attribute.

Attributes	Values
class	integer, real, boolean, name, class_attribute, verb, [verb], program
usage	value, variable
access	directly_represented, generated, named
access-usage	literal, generated_value, named_constant, named_variable
binding	bound, unbound
voice	active, passive
mode	imperative, operator
influence	regular, meta

TABLE 1: Attributes and Values

Dependency	Attributes
name	binding
bound	class, usage
verb	name, voice, mode, influence
operator	class

TABLE 2: Attribute Dependencies

	CLASS							
ACCESS-USAGE	1	2	3	4	5	6	7	8
literal	X	X	X	X	X	X	X	X
generated_value	X	X	X				X	
named_constant								X
named_variable	X	X	X					

where 1: integer, 2: real
3: boolean, 4; name
5: class_attribute, 6: verb
7: [verb], 8: program

TABLE 3: Object Availability

Verb Dictionary

[verb](CREATE_PROG,passive,imperative,meta),literal
[{ SYMTAB <—nul }
name(unbound)$^{(000;)}_{indicant(program)}$
{ SYMTAB <— + || VALUE(indicant)—>
(bound(program,value)) }
program,literal$^{(000.)}_{base}$]

VALUE is an operator that can be applied to objects to yield their value. In CREATE_PROG, VALUE returns the actual name literal used to satisfy the indicant case object requirement. In the CREATE_VAR frame, VALUE is used to yield a class value.

[verb](CREATE_VAR,passive,imperative,regular),literal
[(name(unbound)$_{indicant}$
{ SYMTAB <—SYMTAB + || VALUE(indicant—>
(bound(VALUE(specifier),variable))}$^{+(00,:)}$
class_attribute$_{specifier}$)$^{+(00;0)}_{objective(var/}$]

[verb](COMPOUND,active,imperative,metal),literal
[[verb] (,active,imperative,)$^{+(00;end)}_{objective(begin)}$]

[verb](ASSIGN,active,imperative,regular),literal
[CLASS: ,variable$_{recipient}$
CLASS| {CLASS = real } integer, value$_{donor(:=)}$]

[verb] (BINARY_SELECTION,active,
operator([verb](,active,imperative,)),meta),literal
[boolean,value$_{selector(if)}$
[verb](, active, imperative,)$_{objective(then)}$
[verb](,active,imperative,)$_{objective(else)}$]

[verb] (REPETITION,active,imperative,meta),literal
[boolean,value$_{governor(while)}$
[verb](,active,imperative,)$_{objective(do)}$]

[verb] (ADD,active,operator(CLASS),regular),literal
[CLASS1:integer|real,value$_{objective}$
CLASS2:integer|real,value$_{objective(+)}$
{ CLASS <— integer
(= CLASS1 = real | CLASS2 = real =) CLASS <—real}]

The case frames for the other regular operators in the language are not shown.

Access Form Dictionary

program,literal
< [verb](CREATE_VAR, , ,)$^{op(000;)}$
[verb](COMPOUND, , ,),literal >
CLASS,generated_value
< [verb](,active,operator(CLASS:),) >
CLASS,USAGE,named
< name(bound(CLASS: ,USAGE:)) >

CONCLUSION

Case grammars define PLs in terms of objects and verbs, and their relationships to each other. Although the notation is capable of defining the syntax completely, the emphasis remains at the object rather than at the symbol level. By following CG as a model, we may be able to design PLs that incorporate some features of NL and are therefore more comfortably used. Multiple surface structures can be allowed, perhaps permitting multilingual translators. CGs can also serve as a vehicle for comparing PLs concentrating on their deep representational abilities rather than on their surface structures. Nevertheless, the most important application of CGs offer the following advantages over context-free grammars. First, syntactic details are clustered into punctuation encodings, and can easily be ignored. If errors occur at this level, they are likely to have a minimal effect on the parser's functioning. Second, attribute-value information is stressed, whereas in most context-free grammars and parsers it is ignored. Finally, functional case relationships are emphasized over positional relationships. This suggests that errors of position can be well tolerated.

REFERENCES

[1] C. Fillmore, The case for case. In: *Universals in Linguistic Theory,* E. Bach and R. Harms, Editors, new York, Holt, Rinehart, and Winston, 1968, pp. 1-88.

[2] C.A.R. Hoare and Wirth, N. An axiomatic definition of the programming language Pascal. In: *Acta Informatica,* vol. 2, fasc. 4, 1973, pp. 335-355.

[3] K. Jensen and N. Wirth,*PASCAL User Manual and Report,* New York, Springer-Verlag, 1975.

[4] J. M. Popelas, Ph.D. dissertation, University of North Carolina at Chapel Hill, 1981.

Notes for Contributors

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

Manuscripts should be submitted in triplicate to: Prof. G. Wiechers at:

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.

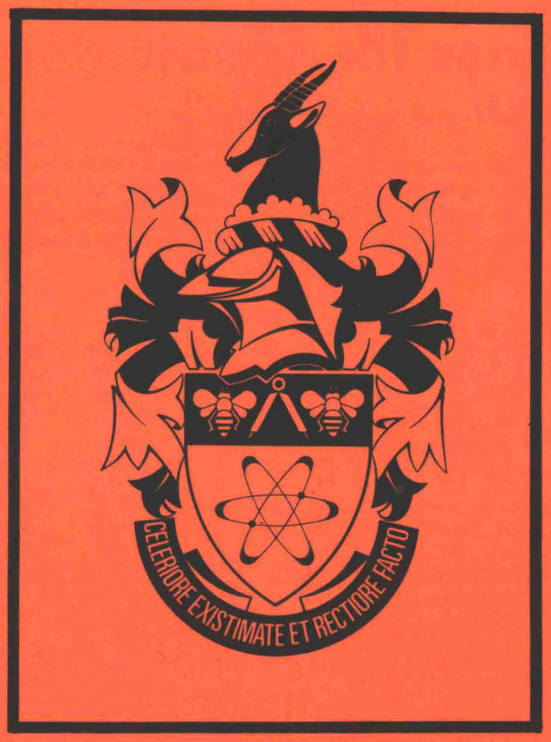
Only original papers will be accepted, and copyright in published papers will be vested in the publisher.

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 second South African Computer Symposium held on 28th and 29th October, 1981.

Printed by Studio Press, Johannesburg