QUAESTIONES INFORMATICAE

Vol. 1 No. 3

March, 1980

Quaestiones Informaticae

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

Editors: Dr. D. S. Henderson,
Vice Chancellor, Rhodes University, Grahamstown, 6140, South africa.
Prof. M. H. Williams,
Department of Computer Science and Applied Maths,
Rhodes University, Grahamstown, 6140, South Africa.

Editorial Advisory Board

PROFESSOR D. W. BARRON
Department of Mathematics
The University
Southampton S09 5NH
England

PROFESSOR K. GREGGOR
Computer Centre
University of Port Elizabeth
Port Elizabeth 6001
South Africa

PROFESSOR K. MACGREGOR
Department of Computer Science
University of Cape Town
Private Bag
Rondebosch 7700
South Africa

PROFESSOR G. R. JOUBERT
Department of Computer Science
University of Natal
King George V Avenue
Durban 4001
South Africa

MR P.P. ROETS
NRIMS
CSIR
P.O. Box 395
PRETORIA 0001
South Africa

PROFESSOR S. H. VON SOLMS
Department of Computer Science
Rand Afrikaans University
Auckland Park
Johannesburg 2001
South Africa

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

MR P. C. PIROW
Graduate School of Business Administration,
University of the Witwatersrand
P.O. Box 31170
Braamfontein 2017
South Africa

Subscriptions

Annual subscriptions are as follows:

	SA	US	UK
Individuals	R2	$3	£1.50
Institutions	R4	$6	£3.00

Quaestiones Informaticae is prepared for publication by SYSTEMS PUBLISHERS (PTY) LTD

for the Computer Society of South Africa.

A High-level Programming Language for Interactive Lisp-like Languages

Stef W. Postma

Department of Computer Science, Rand Afrikaans University, Johannesburg, South Africa

Abstract

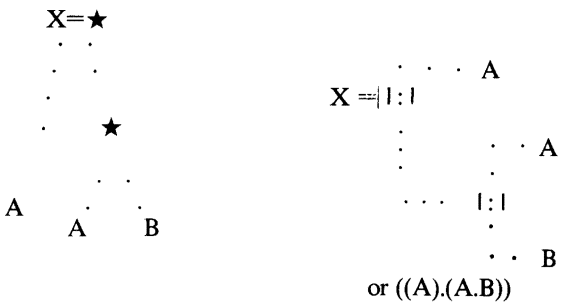
The language LISP 1.5 is neither easy to write nor to read. In the McCarthy et.al. definitive manual on LISP 1.5 a higher level language, MLISP, is used. From the early 1960's various people have proposed higher level languages for LISP. Two types of languages may be distinguished: those based on a LISP programming style, but more readable, and those based on an Algol style. The language PLISP is a language in which it is attempted to reconcile the two approaches to high level languages for LISP. The notation is based on the Algol-style languages but is brief and adapted for terminal use. PLISP allows full access to all the LISP features in a consistent notation. In addition there are features that will appeal to the mathematically mature programmer: full infix notation for not only the assignment, arithmetic, and logical operators, but also for lists, sets, and chained relations. PLISP also has a consistent notation for the first order predicate logic and a variety of ways in which functions may be defined. Top-down program development is facilitated by the use of an assertional language FEA which can be translated to PLISP for test-data generation. Furthermore a definite notation for function stubs is made available — the stubs yield values although the functions called have not been defined at that stage.

1. Introduction: LISP

"... LISP is a formalism for describing algorithms, for writing programs, and for proving properties of algorithms."

John Allen, Anatomy of LISP [1]

The language LISP-acronym for LISt Processor — was defined in the early sixties. Many extensions and modifications to LISP have been proposed over the years, and there is currently no standard for LISP. The language is however, of such beautiful simplicity that a user can find his way quickly in most of the LISP dialects. The power and elegance of LISP makes it an ideal tool for investigating algorithms on a variety of data structures. The basic LISP data type is the leaf-named binary tree — but general graphs may be represented as well. An example of a leaf-named binary tree, X, is:



A LISP function is written in the form:
(function argument ..... argument). The basic LISP functions are:

CAR — mnemonically: Climb-Across-Right
eg. (CAR (CAR X)) is A.

CDR — mnemonically: Climb-Down-Right
eg. (CDR(CDR X)) is B.

CONS — inserts its arguments into a bi-pointer (i.e. bip) element. A bip is the computer equivalent of a node in a binary tree.

QUOTE — used to refer to an item instead of the value of the item. E.g. the value of X is obtained by evaluating:
(CONS (CONS (QUOTE A)())(CONS (QUOTE A) (QUOTE B)))

The basic predicate functions are:

ATOM — to test for elementary items,
e.g. (ATOM (CAR X)) is false
(ATOM (CAR (CAR X))) is true.
If Y is not an ATOM then Y equals (CONS (CAR Y) (CDR Y))

EQ — tests its two arguments for identity.

NULL — tests the NIL (or f) value.
e.g. (NULL (CDR (CAR X))) is true.

COND — The LISP equivalent of the IF statement is actually closer to a CASE construct.

The LISP language not only illustrates most of the issues of computer science, but also is extensively used in Artificial Intelligence and Symbol Manipulation systems. The work and research of Artificial Intelligence is usually either implemented in LISP, or in a LISP-based system, where the specialized A.I. language is translated to LISP and the programs executed by the LISP interpreter.

2. The MLISP Languages

The original definition of the language LISP 1.5 is contained in the LISP 1.5 Programmer's Manual by McCarthy et al [4]. There are however no standards for LISP and the implementations of the language contain either modifications or extensions and therefore define dialects of LISP. LISP is a language that is popular and important in artificial intelligence and in symbol manipulation in spite of the communication difficulties caused by the existence of LISP dialects.

The work reported here was done when the author was registered for a Ph.D. at the University of the Witwatersrand.

A more serious problem is caused by the LISP syntax. A LISP program is written in the form of a list so that programs and data have the same syntax: LISP is a high-level language in which programs can be modified or even computed. The syntax of LISP is such that a LISP program is rather unreadable to a human. Consider the definition of the factorial function:

(DEFN (QUOTE FACTORIAL) (QUOTE (LAMBDA (N)
(COND ((EQUAL N 0)1)
(T(TIMES (FACTORIAL (DIFFERENCE N 1)))))))

The discerning reader may have noticed that the parentheses do not balance! A parenthesis balancing technique that has been implemented is the use of square brackets [4], a right square bracket closes all left parentheses up to and including a left square bracket.

The requirements that must be met to improve the readability of LISP are:

1. Parentheses must be kept to a minimum.
2. Clear distinctions between functions and arguments.
3. A natural way of grouping and indenting statements
4. Clear indication of control functions and blocks

The need for a LISP-based language for human communication was realized from the beginning. A language called MLISP — for meta-LISP- is defined in the McCarthy manual. The rules of MLISP are few but the language never caught on. The MLISP form is easier to read than the original symbolic expression, but it hardly meets the three requirements for readability. Higher level languages for LISP followed one of two lines of development: On the one hand more verbose languages in the style of Algol have been proposed: Words like FOR, DO, WHILE, IF, THEN, ELSE are used for control structures. An early effort is the ALISP of Henneman [2]. A currently available language in Algol style is CLISP of Teitelman. The CLISP language is available in the Interlisp system [11],and is not only a high-level language for LISP but it also incorporates testing and pattern-matching features. The second line of development for high-level LISP languages follows on the MLISP initiative. The control structures are indicated symbolically rather than verbally, but a fair amount of verbiage may still occur. Two examples of these developments will suffice, one by McCarthy [3], in Algol style, the other form a recently published book by Allen [1], in MLISP style.

The function considered is the definition of the EQUAL function for LISP. The EQUAL function yields the value t (i.e. TRUE) if its two arguments have the same structure, and are identical at the lowest (atomic) level.

1. McCarthy: Using **at** for ATOM, **a** for CAR, **d** for CDR

```
equal [x,y] ← if at x
                then if at y then x eq y
                     else F
                else if at y then F
                else if equal [a x, a y]
                     then equal [dx, dy]
                else F.
```

2. Allen:

```
equal [x;y] <= [atom [x] →
                  [atom [y] → eq [x;y]; t → f];
                atom [y] → f;
                equal [car [x]; car [y]]
                  → equal [cdr [x]; cdr [y]];
                t → f]
```

The readability of the above two should be compared to the PLISP example below:

3. PLISP: Using α for ATOM, == for EQ, $\lceil$ for CAR, $\rfloor$ for CDR

```
EQUAL (X;Y) ← [?]
              ? α(X) → [?]
                       ? α(Y) → X == Y;
                       ? ⊤ → ⊥;
                       ?]
              ? α(Y) → ⊥;
              ? EQUAL (⌈X; ⌈Y) → EQUAL
                                 (⌊X; ⌊Y);
              ? ⊤ → ⊥;
              ?]
```

3. A review of PLISP

The language called PLISP (acronym for Publication/Programming LISP) was developed to facilitate communication in LISP. The improved communication is not only between man- and machine (programming) but also human communication of LISP programs (publication.) The symbols used in PLISP are available on any computer terminal that has APL facilities. The language PLISP has otherwise no connection with APL. The restriction of the symbols to the APL character set was suggested by noting the availability of the symbols to the user.

The PLISP notation is as concise as possible, but a degree of redundancy has been built into the language. This is in accordance with Wirth's dictum [13] that languages may —

'Through their very conciseness and lack of redundancy elude our limited intellectual grasp'.

The level of redundancy was found by experimenting with programs written in the various proto-PLISP languages that were designed. The test for acceptance was simple: at the stage where sufficient redundancy was added to make the 'pattern' — the 'layout' — of a program obvious to the user further additions were stopped. The use of indentation and comments form part of the modern techniques of programming, and are provided in PLISP. Moreover, facilities for topdown refinement and program development are provided by provision for program assertions and function stubs that can return values although the functions have not been defined.

LISP is used in two areas: artificial intelligence and symbol manipulation, and the PLISP that is described in this paper has been devised for the symbol manipulating LISP languages. Extensions for the LISP dialects, e.g. QA4 [9] that have been developed for artificial intelligence are being investigated.

The more noticeable features of PLISP are

1. A natural and readable notation for programming. The notation includes the commonly used infix operators, and clearly indicated control-block structures.
2. The control-block structures that consist of more than one statement has each statement in the block clearly marked. These markers are used as prompts by the computer on a terminal system.
3. The functional notation used in programming has been extended to allow for the notations that mathematicians use. Functions may be prefixed or postfixed to their arguments; functional composition is allowed, and a function may be the value of a calculation. The definition of finite functions are also allowed in an almost mathematical notation.
4. Expressions in predicate logic are allowed and are translatable to programs.
5. Data-constants of various types: sets, trees etc.

The relation of PLISP to LISP is quite simple. The semantics of PLISP is given in terms of LISP by means of a simple Syntax Directed Translation Schema [7]. All the LISP functions are

therefore available to the PLISP programmer. In particular no general input and output functions are defined in PLISP since this will vary with the target LISP dialect that may be used.
Planned extensions for PLISP include:

1. An incremental interactive compiler to Lyspe — a LISP dialect with proper I/O and string manipulation features.
2. Extensions to allow for:
   2.1 Features for artificial intelligence;
   2.2 Database facilities;
   2.3 Syntax directed input
3. Program verification and testing based on the assertional language FEA [6].

All the features of PLISP cannot be discussed in this paper but enough constructs are described to give the reader a working knowledge of PLISP. An example of a PLISP program is given in appendix A, and the reader may find it amusing to read through the example before he reads the description of the language — this will enable him to judge the extent to which PLISP succeeds in being not only a programming language but also a language for human communication.
Some terminology has to be defined. The functions that are of type EXPR, or SUBR, i.e. those that receive their arguments in evaluated form are called the EA functions (Evaluated Argument). The functions that are invoked, and then may, or may not, evaluate their arguments are called NEA (Non-Evaluated Argument)-functions.

4. Control Blocks

A control block is required whenever a number of statements are to be considered as a unit. To facilitate interactive programming, and for the sake of consistency, the control blocks of PLISP are all written with the same basic syntax: control block header in which variables may be named for use in the block, the body consisting of one or more statements each preceded by the control block prompt (or marker), and the control block exit. A prompt must be followed by a blank, a colon or a right square bracket.
Nested blocks may have additional identification on their prompts but this is not discussed in this paper.
A PLISP session at the terminal is initiated by the PROGRAM-header: [□ cset-variables] followed by any number of statements which are evaluated immediately after input. After evaluation the computer prompts the next statement by typing □. The end of a session is signalled by the program exit symbol: □]. The cset-variables are only used for exclusively global variables: these variables may not be used as local variables. Values are assigned to them by means of the CSET function. The variables are initialized to the value ⊥ (NIL) at block-entry time. Most programmers will not make use of the CSET variables, but they are required for LISP systems that make use of the APVAL-value of a variable.

```
E.g.                                  Typical translation:
[□A ; B]                              (PROGRAM (A;B)
□: CSET ('A;5);                       (CSET (QUOTE A) 5)
□: CSET ('B;Ao oA);                   (CSET(QUOTE B)(CONS A A))
□: □B;                                (PUTDATA B)
□]                                    )
```

The other control structures are:

1. The CONDitional corresponding to the case statement.
2. The PROG corresponding to BEGIN . . . . . END.
3. The REPEAT corresponding to various kinds of loops.

Local variables may be named in each of the control structures. Such variables are relatively global to any contained blocks, unless they have been named again. The local variables are always initialized: if at block entry time there is a relatively global variable with the same name then the local variable is initialized to the global value, otherwise the local variable is initialized to ⊥(NIL). The values of the local variables are lost at exit from the block.
The conditional, with its prompts, is of the form:

```
[? local variables]
? p1 → e1;
:
:
? pn → en;
?]

E.g. [? A;B]
     ? 'A: = ⌊ B → ⌈ B;
     ? ⌊ A → ⌈ ⌊ B;
     ? ⊤ → B;
     ?]
```

which translates to:

```
(COND (A;B) ((SET (QUOTE A) (CDR B))
  (CAR B)) ((CDR A)(CADR B)) (T B ))
```

or, if the target LISP does not allow local variables in a COND, the translation is:

```
(PROG (A;B) (COND ((SET (QUOTE A) . . .
```

If none of the pi have a non-nil value then the conditional is, typical LISP-language like, undefined.
The PROG, with its prompts, is written:

```
[ω local variables]
ω statement;
:
:
ω statement;
ω]
```

The statements of a PROG are evaluated from the first statement to the last statement (i.e. from alpha to omega). The value of the last statement is the value of the PROG — hence the mnemonic symbol ω (omega) used as the PROG symbol. Labels and a GO-function are not defined in PLISP. If however they are really required, the LISP coding may be used in the PLISP program to achieve the desired effect. LISP coding may be entered at any point by enclosing that coding in the special quotingbracket: slash (/).
The repeat (loop) construct must have an exitcase as one of its statements. The exitcase determines the value of the repeat, and replaces the COND with a RETURN of LISP. Furthermore, the exitcase acts like the COND in a PROG of LISP: If no condition of the exitcase is satisfied then the evaluation continues with the statement following the exitcase.
A repeat block is written:

```
[ρ local variables]
ρ statement - 1;
:
:
ρ statement -n;
ρ [ε local variables]
   ε p1 → e1;
     :
     :
   ε pj → ej;
   ε]
ρ statement -a;
:
:
ρ statement -z;
ρ]
```

Either or both of the sequences of statements 1-n or a-z may be empty.

5. Infix Operators and Data-Types

It is customary to assign a precedence among infix operators, for example a + b × c is usually interpreted as a + (b×c). An operator furthermore is usually interpreted as being left associative, i.e. a+b+c is evaluated as (a+b)+c. In PLISP the operators are classified as:

LA — left associative eg. A+B+C is (A+B)+C
RA — right associative e.g. A:=B:=C is A:=(B:=C)
MLA — mutually LA e.g. A-B+C-D is ((A-B)+C)-D
MRA — mutually RA e.g. $A_{oo}B_oC_{oo}D$ is $A_{oo}(B_o(C_{oo}D))$
LAC — LA-chain e.g. A =B = C is (A= B) ∧ (B = C)

The operators and typical data constants are given in the table below. The entries have low precedence at the top to high precedence at the bottom.

Infix Operators

Operator	Type	Lisp-name	Data constant e.g.
←		DEFUN	
:=	RA	SET	
<=	RA	SETSTRING	··String const··
oo	RA	CONS	
o	RA	CONC, APPEND	/S-expression/
ρ	RA	PAIR	
~	RA	NOT	
∧, ∨	MLA	AND, OR	⊥ is 'false'
⊃	LA	IMPLIES	T is 'true'
⊃⊂	LAC	EQUIV	
<,>,=, ≠, ε, ⊂,			
= =,	MLAC	EQ	
/ /,		SUBSTRINGP	
⊂⊃		SUBSETP	
∩,∪	MLA	INTERSECTION, UNION	
¯	LA	SETCOMPL, relative	
★	LA	CARTESIAN	{X predicate}
2★	RA	POWERSET	{X ε S predicate}
+,-	MLA		integer
× ÷	MLA		fixedpoint
↑			floating point
	LA	CATENATE	
↓	LA	SUBSCRIPT	
=>	LA	INDIRECT addressing	
	RA	CDRASSOC	≤edge-named tree≥

6. Prefix, Infix, and Suffix Functions

Three types of function names may be used in PLISP. Firstly any LISP function may be used, secondly a number of standard LISP functions are represented by symbols, and finally the user can define his own functions (and operators). The defined functions may have any valid name, or the name may be formed from a prescribed set of symbols. The predefined function symbols are all used as prefix functions. If they take a single argument then that argument need not be enclosed in parentheses if the argument is an identifier. The predefined function symbols are:

E.g.	translates to:
'A	(QUOTE A)
⌈ B	(CAR B)
⌊ C	(CDR C)
□ E	(PUTDATA E)
αF	(ATOM F)
⊥ G	(NULL G)
εαH	(EVALA H)
ρA(I;J)	(RPLACA I J)
ρ D(K;L)	(RPLACD K L)

The function defining functions are:
ΔE (fname genlambda) for EA functions
ΔN (fname genlambda) for NEA functions
ΔT (fname genlambda) for temporary EA functions

Function calls may be specified in various ways. The first type of call — symbolic function followed by identifier-argument — is given above. The most common type of function call is of course: function-name (arguments) ⌊
which is also given above. A number of other ways of invoking a function are also allowed, they are:

1. Post-fixed function calls
2. Compound function calls
3. Calculated function calls
4. Tabular function definition
5. Generalized lambda functions

In a post-fixed function call the arguments precede the function. In this case the arguments are enclosed in a special pair of brackets, i.e.: [(arguments)] function-name.

A compound function call is the PLISP equivalent of the mathematical composition of functions. The form F(B(H . . . (M(args). . .) may be written as: [o F ; G; H ; . . . ; M o] (args)

A function is represented by a lambda-expression in the LISP languages. A function name is used by the EVAL-routine to establish the particular lambda-expression that defines the function. In LISP therefore a function — be it name or lambda-expression — may be calculated. Since a calculation may be any form a special notation is needed to distinguish the calculated functions from other forms. A calculated-function call is therefore written in the form:

[ιexpressionι] (args)

A function is always defined, even if only implicitly (in PLISP that is) in terms of a lambda-expression. A lambda expression is specified by: [ν proto-arguments] expression ν]. The proto-arguments are the identifiers that are used as local variables in the expression. The proto-arguments take on the values of the arguments on the systems a-list at function invocation. They are either simple identifiers, or are identifiers prefixed with an iota-symbol — the latter are called inhibited proto-arguments. The inhibited proto-arguments are not stacked but only modified on the systems a-list if an inhibited recursive function call is encountered at time of evaluation. An inhibited function call is specified by prefixing the function name with an iota. This allows recursive functions to be evaluated wholly or partially iteratively.

The tabular function definitions are used to define EA functions according to standard mathematical notation. An example suffices:

ADD5: INTEGER ★ INTEGER → INTEGER:
[★0 1 2 3 4
★0→0 1 2 3 4
★1→1 2 3 4 0
★2→2 3 4 0 1
★3→3 4 0 1 2
★4→4 0 1 2 3 ★]

Expressions in the first-order predicate logic are also interpreted as implicit lambda expressions. The functions corresponding to these expressions take only sets or integers as arguments. The expressions allowed are quantified formulae, and the first quantifier must be a restricted quantifier [8], which specifies the universe of discourse. The other quantifiers may be restricted or not depending on whether they refer to another or the previously specified universe.

The unrestricted quantifiers are:
[∧ X] i.e. for all X
[∨ Y] for some Y
[∨1 Z] for one and only one Z
The restricted quantifiers:
for sets: [∧ X∈S] i.e. for all X of S
[∨ Y∈T] for some Y of T
[∨1 Z∈U] for one and only one Z of U
for integers: [∧ X < N] for all X less than N, X ≥ 0
[∨ Y < M] for some Y less than M, Y ≥ 0
[∨1Z <P] for one and only one Z less than P, Z ≥ 0.
E.g. [∧ X ∈ S] [∧ Y] [∨ W ∈T] P(XYW) ∧]
In the above expression X and Y range over the same universe S and W ranges over the set T. The quantifier expression is translated to a lambda-expression with S and T as proto-arguments. In the body of the lambda expression, the X,Y and W are defined as local variables. During execution of the function, X and Y will range over the set which is the argument corresponding to S, and W over the set which is the argument corresponding to T.

7. Top-Down Development: Assertions and Stubs

There are three requirements for top-down programming: the first is a language facility to enable the user to express overall, or top-level, program constructs. The second is a facility for defining program or function stubs: procedure calls to routines that are yet to be written. The third is a language facility to enable the programmer to make statements or assertions about the program.
Assertions may be used in PLISP programs in addition to comments. The body of the assertions are written in a language called FEA which has been described elsewhere [6]. The FEA-assertions are English-like statements in the predicate logic that can be translated to PLISP, and may be used as input to a program verifier.
An assertion is written in the form:

[α local variables]
α FEA assertion
:
α FEA assertion
α]

Function stubs are calls to functions that have not yet been defined. The called function, until it is defined, usually does only some simple-minded thing like return a constant value or prints out a message e.g. 'FN AX3 CALLED'. The function stubs that are possible in LISP languages can be much more powerful since an identifier can have a value and also be defined as a function. The FNSTUB function then tests its function argument to see whether it has been defined as a function, if so FNSTUB calls that function; if the function of the functionstub has not yet been defined, then it must have a list of values as its value. The FNSTUB returns the CAR of the list as the value of the functionstub call, and sets the list of values to the CDR of the list.
In PLISP a functionstub is written:

[Δ id: id; . . . ; id/id : id; . . . ; id/ . . .]
optional assertion
function (arguments)
Δ]

The - id: id; . . .; id - are declarations and may be used in any way by the programmer. For example, data types may be declared, or variables may be declared as global in the function to be defined.

References

[1] ALLEN, J.R. (1978). *Anatomy of LISP,* McGraw-Hill, New York.
[2] HENNEMAN, W. (1964). An Auxiliary Language for more Natural Expression — The A-Language, in Berkeley, E.C. and Bobrow, D.G. (Eds). *The Programming Language LISP : Its Operation and Applications,* M.I.T. Press, Cambridge, Mass.
[3] McCARTHY, J. (1977). Another SAMEFRINGE, *SIGART Newsletter,* No. 61, Feb. 1977, p.4.
[4] McCARTHY, J., et. al. (1965). *LISP 1.5 Programmer's Manual,* 2nd ed., M.I.T. Press, Cambridge, Mass.
[5] MANNA, Z. and WALDINGER, R. (1977). *Studies in Automatic Programming Logic,* North-Holland, New York.
[6] POSTMA, S.W. (1978). FEA — A Formal English Subset for Algebra/Assertions, *SIGPLAN Notices,* **13,** 7, 43 - 59.
[7] POSTMA, S.W. (1977). Some Useful Techniques for Defining Formal Languages — With examples from FEA and PLISP, paper read at 1977 Symposium of the Computer Society of South Africa
[8] ROSSER, J.B. (1953). *Logic for Mathematicians,* McGraw-Hill, New York.
[9] SAMMET, J.E. (1969). *Programming Languages: History and Fundamentals,* Prentice-Hall, Englewood Cliffs.
[10] SANDEWALL, E. (1978). Programming in an Interactive Environment: The 'LISP' Experience, *ACM Computing Surveys,* **10,** 1, 35 - 71.
[11] TEITELMAN, W. (1975). *Interlisp Reference Manual,* XEROX, Palo Alto.
[12] UNIVERSITY OF WISCONSIN COMPUTING CENTER, *LISP Reference Manual for the UNIVAC 1108,* UPLI Reference No. 800022.
[13] WIRTH, N. (1974). On the Design of Programming Languages, in *Information Processing 74,* North-Holland, Amsterdam, pp. 386-393.

Appendix A

```
*** Definition of the AND function. Symbols used: ΔN -
  DEFNEA;
* ΔE — DEFEXPR; ∇ — LAMBDA; ⊥ — NULL, NIL; ⊤ —
  TRUE; εα — EVALA;
* ι — Inhibitor, ⌜ — CAR; ⌞ — CDR; ' — QUOTE.
* AND is a NEA, LA function, and it is the PLISP equivalent of
* the logical conjunctive. If it has no arguments then its value is ⊤.
*    If it has one argument then the value of that argument is
* a list of items to be ANDed. Else its arguments are evaluated
* from the left. ***
ΔN('AND '[∇PA]
[?] ? ⊥PA →⊤;
    ? ⊥(⌞PA)→ ANDEA(εαPA:⊤);
    ?]Δ])
    ΔE ('ANDEA '[∇ ιPB; ιPC] *** Auxiliary Function ***
    [?] ? ⊥PB→⊤;
    ? PC →[?] ? [PB →ANDEA(⌞PB;PC);
              ?⊤→⊥;
              ?]
    ? ⊤→[?] ? εα(⌜PB)→ιANDEA(⌞PB;PC);
              ? ⊤→⊥;
              ?]
    ?]∇]
    *** alternative definition using iteration ***
    ΔN ('AND '[∇PA]
          [?] ? ⊥PA→T;
              ? ⊥(⌞PA)→[ω]ω'PA:=εαPA;
                        ω[ρ] ρ[ε] ε ⊥PA→⊤;
                                 ε~(⌞PA)→⊥;
                                 ε]
                             ρ'PA:=⌞PA;
                             ρ]
                        ω];
                ? ⊤→[ρ]ρ[ε] ε ⊥PA→⊤;
                           ε~(εα(⌜PA))→⊥;
                           ε]
                      ρ 'PA:=⌞PA:
                      ρ]
                ?]∇])
```

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 or 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 either Dr. D.S. Henderson or Prof. M. H. Williams at

Rhodes University
Grahamstown 6140
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

Part 2 of the proceedings of the first South African Computer Symposium on Research in Theory, Software, Hardware, organised by The Research Symposium Organising Committee of The Computer Society of South Africa. 4 & 5 September 1979, Pretoria.

Contents/Inhoud