QI QUAESTIONES INFORMATICAE

Volume 4 Number 2 May 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-Africa 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-Africa en van die Suid-Afrikaanse Instituut van Rekenaarwetenskaplikes

Subscriptions

Annual subscription 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.

A PERSISTENT PROGRAMMING LANGUAGE - PRELIMINARY REPORT

Sonia Berman
University of Cape Town
Rondebosch 7700

A persistent programming language PPL is currently being designed at the University of Cape Town. The language treats persistence (permanent storage) as an orthogonal property of data and is consistent in its treatment of persistent and transient data. The persistence of an object is the length of time the object exists. In traditional programming languages, data does not last longer than the execution of the program unless some storage agency such as a file or database is used. With persistent programming, data can continue to exist after programs have been run to completion, and the method of accessing the data is the same for both long and short term objects. The inclusion of semantic data modelling concepts [8] in a programming language is also being investigated in PPL. This paper describes the need for a persistent programming language and gives an overview of PPL.

1. MOTIVATION

Currently, software design is complicated by the need to perform two mappings: from real world to program model and then from program to stored model. It is clearly wasteful if different code has to be written and maintained to achieve the same effects - in the one case on persistent data, and in the other on transient data. Furthermore the program and stored models have conflicting interfaces with different approaches to data structures and operators. This is largely a result of the independent development of file-handling and database systems on the one hand and programming languages on the other. Subschemas are necessary in database systems to perform translations between names and base types, to make those in the database useable in the language. The PPL language is an attempt to rectify this situation by providing an integrated programming enviroment for the handling of both temporary and permanent data in a uniform manner. The language should reduce the effort and complexity involved in defining and manipulating data thereby allowing faster program development and easier maintenance

2. DATA TYPES

In PPL the basic data types are strings, integers, reals, scalars, arrays, sets and aggregates [9]. The maximum length of a string must be specified and the size of an integer or real can also optionally be given. Aggregates are composed of other objects that can be of any type, much like Pascal records. A set is analogous to a Pascal set except that any element type is permissible. The only difference between this and an array is that the latter is of predefined size and is ordered, i.e. the i'th element of an array can be referenced, but there is no notion of the i'th element of a set. A persistent set is equivalent to a relation; a persistent array similar to a file. Since aggregate, array and set elements can be of any type several complex structures are possible such as sets of aggregates, sets of sets, sets of arrays and arrays whose elements are sets of aggregates / sets / arrays, etc. Special modalities such as text, audio or image data can be catered for by defining them as strings. In PPL any variable can be defined as permanent, not only files or relations as is the case with present systems. In this way, if a database contains items that are given unique numbers, a permanent integer variable ITEMNO can store the last value used. Similarly, data of any type can be persistent. The scalar type therefore enables a database definition to include values to be used for MARRIED, SINGLE, WIDOWED and DIVORCED for example. Names can also be associated with specific values of integer, string, real or scalar to make the program code easier to follow. Thus if the first digit of a 6-digit Stylenumber designates its department, then the name LADIES can be associated with values 100000 to 199999, MENS with 200000 to 299999 and so on.

Since arrays and sets can comprise elements of any type, a great deal of flexibility and versatility exists, and high level types can be introduced and denoted succinctly. PPL is conceptually simpler and more natural to use than existing file-handling or database systems as it

is object-oriented rather than record-oriented and deals with logical units such as arrays and sets rather than physical structures like files. The programmer's view of an aggregate is not necessarily identical to the record or tuple structure(s) that represent it. What the programmer conceptualises as a single aggregate may in fact be represented by more than one type of record or tuple.

"There is a concensus that classes of objects and type hierarchies are useful" [2]. The concept of type hierarchies or generalisation / specialisation [9] is incorporated in PPL. Specialisation enables a new data type to be defined based on any existing data type. Examples: an EMPLOYEE aggregate based on a PERSON aggregate, a DATE based on an integer. The components of EMPLOYEE would specify additional attributes in addition to those of ordinary PERSONs. Thus if PERSON had three components (eg. Name, Age and Sex) and EMPLOYEE had four components (eg. Dept, Projects, Job, Wage) then EMPLOYEES would have altogether seven components. Inherited components can be redefined if necessary. For example if EMPLOYEE also had a component Age then this might have been declared in order to specify a constraint on an employee's age (such as Age > 15 and Age < 66) which does not apply to PERSONs in general. The declaration of a specialised type can include a predicate that determines whether or not an instance of the general object is also an instance of that specialised type. For example an EMPLOYEE in a university database can be defined as "isa PERSON : Employer = 'UCT' ". This ensures that a PERSON occurrence is not classified as an employee and given values for EMPLOYEE components unless its Employer value is correct. Virtual data can be defined by following its declaration with the formula to use in computing its value. This formula can employ any PPL operator (depending on the context) along with constants and/or other components in the same aggregate. As an example, an aggregate that is a subset of another aggregate can be defined using a conditional expression, as in Weakstudents EQUALS Students(Students.grade < 50). As an example of numeric virtual data, a student's average which is computed from several results can be defined as say Average EQUALS (Prac*2 + Test + Exam*3) / 6. Aggregate components which are themselves aggregates are designated either "weak " or "strong" (the default). If weak, a deletion or update of the aggregate of which it is a part will be automatically followed by a deletion or update of the corresponding component aggregate. If a STUDENT aggregate has a property Marks-Gained of type MARK it should be declared WEAK to ensure that the MARK objects for that student are erased if the student is deleted, as they are no longer of interest.

Predicates can be specified for variables and data types such as DATE or SALARY to define integrity or security constraints. Most programming languages have abrogated responsibility for protection, leaving it to the operating system which controls data with less precision. Thus read/write keys can at best be attached to whole files; in PPL they can be attached to any variable (or aggregate component) - even a simple integer. Integrity assertions can be specified for complex as well as simple data types to stipulate relationships that must hold between the components of an aggregate or members of a set or array. PPL permits integrity specifications with either data or transactions so that the programmer can choose whichever method is natural to him. This flexibility should not complicate the language, as the same syntax is used in both cases. Integrity assertions in PPL include a means of specifying functional dependencies [3] that hold between the components of an aggregate. For example Partnumber -> Price stipulates that each partnumber can be associated with only one price. Thus if there are several aggregates of this type in a set or array the system must ensure that wherever any two agree on partnumber, they also agree on price. The keys of an aggregate can be defined: These define components which can act as identifiers for the aggregate; i.e. components whose values are unique over all such aggregates. For example Name and Regnumber might be alternate keys for STUDENT. It should be noted that persistent data will have its definition stored with it, to enable the checking of both a program's declarations and its behaviour, to ensure consistent interpretation of the data throughout the system. This description is analagous to the schema facility in a database environment.

There are several problems and limitations of record-based systems such as files and databases [7]. These stem from the fact that it is not easy to reconcile records with entities or relationships in the real world. A record often stores data on more than one entity (eg. an EMPLOYEE record also contains information about spouse/children) and an entity is often represented by several records (one record cannot always contain all the facts about an entity, as a relationship between two entities will not be stored with both entities, and may be stored with

neither if it is a many:many mapping!) Relationships can be represented in several ways such as by a seperate record type, within the record for one or more of its participating entities, or in another record. Often the fields of a record do not apply to all individuals (this phenomonen is handled in some convential programming languages like Pascal through the use of variant records). In other situations the same field can reference different kinds of object in different circumstances. Such a situation arises for example if a Project-Controller field may either refer to an employee or a department, depending on the project.

The above-mentioned problems are handled by PPL in the following way. An entity or relationship is represented by an aggregate. Since this is an abstract data type, rather than a physical unit, its implementation as one or more records is transparent to the programmer. Thus if a conceptual entity is physically represented by several records this will not in any way affect how that entity (aggregate) is handled in the program. This should be contrasted with conventional file-handling and database systems where the knowledge of the underlying record structure is essential if one is to use the data at all. If an aggregate contains data about another entity (eg spouse in EMPLOYEE) then this fact is highlighted in the EMPLOYEE definition by specifying that component as "Spouse : PERSON;". Whereas a record for an EMPLOYEE in a conventional system could contain fields Name, Age, Spousename, Spouseage the aggregate in PPL would have components Name, Age and Spouse so that the existence of the sub-aggregate/entity is immediately apparent. Specialisation solves the problem caused by fields which are applicable to certain occurences of an entity type only, rather than to all such entities. For example if the field Spouse is only applicable to married employees it would not be included in the EMPLOYEE aggregate; instead a special type MARRIED-EMP would be defined to contain this field and any other data relevant only to such employees. A field such as Project-Controller that can refer to two different types of object can be catered for by defining a type Controller and declaring Employee isa Controller and Department isa Controller.

3. PROGRAM STATEMENTS

PPL provides for traditional assignment, alternative and iterative statements using a Pascal-like syntax. The PPL notation has been designed to be as concise as possible, with minimal punctuation, yet easy to read and understand. Unlike most other languages, there is a notation for literals of all types such as [element, ..., element] for arrays, { element, ..., element} for sets and <name:value, ..., name:value > for aggregates.

Examples:

```
matrix [1..3] = [0, 3+j, i*2];
setx = {0, 3+j, i*2};
setc = { <size:30, name:"CS3>, <name:"ITC">, <name:"MSC",size:3>}
```

The last example stores three agregates in a set. Note that components are referenced by name, not by position; that one or more can be omitted and that they can be specified in any order. The generic type of objects in any operation is checked, to ensure for example that ship-size(an integer) and arrival-date(a date) are not compared (which would be possible in other languages since DATE isa INTEGER.) COMMIT ... END is used to identify the beginning and end of a logical processing unit or transaction which manipulates permanent data; in contrast to BEGIN ... END which delimits a group of statements affecting only transient variables. This is necessary to enable concurrency and rollback to be handled correctly. Exceptions can be defined which specify an (error) condition and the actions to take if it arises. All statements can be applied to either persistent or transient data, except for the FREE statement which deletes a variable or element.

The operators of the language include all the arithmetic and logical operators and functions available in Pascal (from which PPL is derived) as well as exponentiation for which the symbol ** is used. Set operations for union, intersection, difference and Cartesian product are denoted by + / - and * respectively. String and scalar functions are SUCC(essor), PRED(ecessor) and ORD(inal). The symbol + can also be used with strings or arrays to denote concatenation. (Overloading of symbols like + above is recommended by authors such as Horowitz [6] because it supports the use of natural and conventional notations, allowing the same operator to be used in

different contexts in a system without confusion. There is no risk of ambiguity as the particular operation required can always be uniquely determined from the context in which the symbol is used.) Several elements of an array can be referenced directly by specifying a range of (consecutive) elements, as in

```
matrix[ i .. j-3 ] := 0;
```

A set or array name can be followed by a parenthesized expression to reference several selected elements. As an example PARTS(PARTS.price > 10) references those aggregates in the PARTS set or array of which the Price component exceeds 10. This is equivalent to retrieving all tuples of a relation (or all records of a file) for which the retrieval condition "price > 10" holds. Sets can be used as the control of a loop iterated once per member (here a sort order can be specified). The use of a loop was considered preferable to a similar construct in PS-Algol [1] which provides a procedure that will apply another procedure to every member, or until the procedure returns FALSE. This approach is essentially a special case of the mechanism used in PPL, where the loop body consists of a procedure call. The set operations of the language can reference set variables, {} (the empty set), enumerated sets such as {"Smith", "Jones", "Doe"} or the universal set denoted by the reserved word USET.

Modules are treated as first class objects and hence can have persistence as well. In this way the events or activities in the enviroment are treated in the same way as concrete objects. Modules can accept any data type as parameter and can return any type of value. Thus recursive handling of complex types is also possible. Declarations of persistent data is allowed within modules so that libraries can be defined for manipulating such data, keeping their implementation details transparent. The implementation can later be replaced by a more efficient one without affecting the rest of the system.

PPL exhibits complete data independence. This means that the users of data are completely unaware of its physical representation and are immune to changes in the physical organisation. The language should be easy to use and in particular the manipulation of data at the logical rather than physical level (eg. aggregates and sets not records and files or pointers) means that operations are more high-level and simpler to employ correctly.

Finally, it should be noted that PPL is relationally complete [3]. This criterion implies that any permanent data can be accessed using only one statement; there is never a need for procedurality in retrieving specific information. To prove relational completeness, one need only show that the relational algebra operators "join" and "project" can be simulated in the language [3]. In PPL projection is denoted by "set.component". A join of two aggregates on a common component is achieved using `set1 * set2 ( set1.common1 = set2.common2 )`.This takes the cross-product of the two sets and then selects those members where the common components have matching values, giving the exact effect of a join.

A syntax specification for PPL appears in the appendix.

4. PROGRAM EXAMPLE

This section presents an example to illustrate all the salient points of PPL. It must of neccessity be small enough to be understood, yet non-trivial. The example chosen is that used by Atkinson [2] in his chapter on assessing the progress of persistent programming. The example indicates the data description necessary to represent the family trees of a set of people. It demonstrates both a straight-forward and a complex, computed retrieval as well as an update. This program should show that PPL enables simple things to be done simply and complex tasks to be programmed in an easy, succint and readable manner.

```
MODULE MAIN;
TYPE sextype : (male, female);
     date : AGGREGATE
          day : INTEGER CHECK  dateok (day, mth, year) =1;
          mth : INTEGER CHECK (mth>0) AND (mth <13);
          year : INTEGER 4 CHECK (year>1984) OR (year=0)
        END;
     marriage : AGGREGATE
          man : PERSON CHECK man.sex = male;
          woman : PERSON CHECK woman.sex = female;
          from, upto : date
        END;
     person : AGGREGATE
           name : string 30 KEY;
           sex : sextype;
           birth, death : date
              NAMES alive : death.year = 0,
                    dead : death.year <>0;
           father, mother : person;
           child, spouse : SET OF person;
           marriages : SET OF marriage
         END;
      people : SET OF person;

PERM humans : people;
VAR ancestors, gen : people;
BEGIN
     COMMIT (* find all females :- *)
            WHILE humans (humans.sex = female) DO
                  WRITE (humans.name)
     END;

     COMMIT (* find all ancestors of Joe Doe :- *)
            ancestors := {};
            gen := humans (humans.name = "Joe Doe");
            WHILE gen <> {} DO
            BEGIN gen := prevgen(gen);
                  ancestors := ancestors + gen
            END;
            WHILE ancestors DO
                  WRITE (ancestors.name)
      END;
END;  (* main *)

PERM MODULE prevgen (USES gen,tree : people) : people;
 (* given a generation, finds preceding generation *)
BEGIN
     prevgen := tree (tree.child / gen <> {} )
     (* recall / is intersection *)
END;

PERM MODULE addbaby (USES baby:person; VAR mom,pop:person;
                     VAR tree:people);
BEGIN
     IF (mom.sex <> female) OR mom.dead OR
        (pop.sex <> male) OR pop.dead :
        WRITE ("ERROR")
     ELSE
         COMMIT tree := tree + {baby};
                mom.child := mom.child + {baby};
                pop.child := pop.child + {baby}
         END
END;
```

5. CONCLUSION

The language design was divided into three phases : conceptual formulation, specification and implementation; of which only the first two have been completed in the current version. A syntax checker has been written and work on a compiler is currently in progress. Implementation of the language will require the design of a store for each data type. As transient variables can be accomodated in the same way as they are at present in conventional programming languages, it is the treatment of persistent data which is of interest here. It is envisaged that sets will be represented by relations, arrays by direct-access files and permanent variables of type real / integer / string / scalar stored together in a seperate file (except for large string variables which will constitute a seperate text file each). Relations are analogous to (persistent) sets as their elements are of the same type, have no ordering defined, contain no duplicates and can be manipulated using traditional set operations as in Codd's relational algebra language [4]. Aggregates would be implemented as records. Hence arrays of aggregates would become files of records and sets of aggregates would be relations of records (tuples). Finally, for each aggregate type a file would be created to store persistent variables of that type.

PPL is presently a conceptual language and several issues remain to be investigated, such as providing for different user views of data, a browsing facility for permanent data and a means of specifying physical access methods. The primitives of the language must still be assessed for adequacy and completeness, and consistency rules such as referential integrity must be developed for them. Referential integrity [5] specifies that an aggregate cannot exist unless its component aggregates exist. As an example of the latter, if aggregate ENROLLMENT has components date, accepted-by, studentname and coursename (where the last two are aggregates themselves) then an ENROLLMENT should not be created if the STUDENT and COURSE to which it refers do not exist. It is intended that the primitives of the language be formally defined in terms of set theory or predicate logic so that formal data design and program verifiers can be developed.

PPL is a language which treats all data uniformly and consistently; in particular persistence is an orrthogonal property of objects and semanticaly rich definitions are possible. Permanent data is represented using logical, not physical, primitives. A particular strength of PPL is the small number of basic concepts (types and operations) underlying the language. It has introduced new programming power in alllowing succint expression defining computation on large / infinite sets. The language attempts to demonstrate that the traditional data model based on types which are easy to engineer in a computer (ie. records/aggregates and arrays) can form an adequate model for data outside as well as within programs. Ultimately all database concepts will be embedded within the semantics and notation of the programming language and will rarely be consciously used. That is the goal of persistent programming research.

REFERENCES

[1] Atkinson,M.P., Chisholm,K.J. and Cockshott,W.P., "PS_Algol: An Algol with a Persistent Heap" *ACM SIGPLAN Notices* 17(7), 1981.

[2] Atkinson,M.P. et al, "Progress with Persistent Programming", in Database - Role and Structure", Cambridge University Press, 1984.

[3] Codd,E.F., "A Relational Model for Large Shared Databanks", *Comm. of the ACM* 13(6), 1970, pp. 377 - 387.

[4] Codd,E.F., "Relational Completeness of Data Base Sublanguages" in Data Base Systems, Courant Computer Science Symposia Series, Vol. 6, Englewood Cliffs, N.J., Prentice-Hall, 1972.

[5] Date,C.J., "Referential Integrity", in Proc. of the seventh International Conf. on Very Large Databases, Cannes, France, 1981.

[6] Horowitz,E., "Fundamentals of Programming Languages", Springer-Verlag New York, 1983.

[7] Kent,W. "Limitations of Record-Based Information Models", *ACM Trans. on Database Systems* 4(1), 1979, pp. 107 - 131.

[8] Kershberg,L., Klug,A. and Tsichritzis,D.C., "A Taxonomy of Data Models", in Systems for Large Databases, Lockeman P.C. and Neuhold, E.J. ed.s, North-Holland 1976.

[9] Smith,J.M. and Smith,D.C.P., "Database Abstractions: Aggregation and Generalization", in *ACM Trans. on Database Systems* 2(2), 1977, pp. 105 - 133.

APPENDIX - PPL SPECIFICATION

```
program     ::= module { module }

module      ::= [ "PERM" ] "MODULE" identifier
                [ ( arguments ) ] [ : typeid ] ;
                [ declarations ] block ;

arguments   ::= paramlist {; paramlist }

paramlist   ::= "VAR" parameters | "USES" parameters

parameters  ::= decllist ; { decllist ; }

decllist    ::= idlist : identifier

idlist      ::= identifier { , identifier }

declarations ::= [constants] [types] [variables] [perms]

constants   ::= "CONSTANT" const ; { const ;}

types       ::= "TYPE" typedef ; { typedef ; }

variables   ::= "VAR" varlist ; { varlist ; }

perms       ::= "PERM" varlist ; { varlist ; }

const       ::= idlist : value

typedef     ::= identifier : typedec1 { semantics }

typedec1    ::= "INTEGER" [integer] | "REAL" [integer] |
                "STRING" [ integer ] | scalar | agg | arr | set | identifier

agg         ::= "AGGREGATE" { isa } aggdef [ fds ] "END"

aggdef      ::= fieldlist ; {fieldlist ;}

fieldlist   ::= decllist {fieldspec}

fieldspec   ::= { semantics } | "KEY" | "WEAK"

fds         ::= "FDS" { idlist "->" idlist; }

set         ::= "SET OF" identifier

scalar      ::= ( idlist )

arr         ::= "ARRAY" [ dim { , dim} ] "OF" identifier

dim         ::= value ".." value

isa         ::= "ISA" def { , def}

def         ::= identifier [ : expression ]

semantics   ::= checks | passwds | equals | names

names       ::= NAMES namedef { , namedef }

namedef     ::= identifier : expression

checks      ::= "CHECK" expression [ "ERROR" execpart ]

passwds     ::= "READKEY" string | "WRITEKEY" string

equals      ::= "EQUALS" expression

block       ::= execpart [ "CHECK" expression "ERROR" execpart ] ;

execpart    ::= statement |
                "BEGIN" statement ;{ statement ;} "END" |
                "COMMIT" statement ;{ statement ;} "END"
```

```
statement   ::=  asg | in | out | cond | loop | "BREAK" |
                 identifier "(" arglist ")" |
                 "RETURN" "(" expression ")" | "FREE" var

arglist     ::= expression { , expression }

asg         ::= var := expression

in          ::= "READ" ( inlist )

inlist      ::= [ identifier , ] rdval { , rdval }

out         ::= "WRITE" ( outlist )

outlist     ::= [ identifier , ] wval { , wval }

rdval       ::= var | /

wval        ::= expression [ : format ] | /

format      ::= integer [ : integer ]

cond        ::= IF alternative { ; alternative} [ ELSE block ]

alternative ::= expression : block

loop        ::=  while | for | repeat

while       ::= "WHILE" expression "DO"
                [ "ORDER" idlist ] block

for         ::= "FOR" var := expression "TO" expression
                "BY" expression "DO" block

repeat      ::= "REPEAT" statement; { statement; }
                "UNTIL" expression
```

The definition of an expression and identifier is the same as in Pascal, except for the specification of a variable name and value.

```
var         ::= varb1 |
                varb1 [ sub ( , sub } ] |
                varb1 ( expression ) |
                varb1 [ sub { , sub } ] ( expression )

varb1       ::= identifier {"." identifier}

sub         ::= expression [".." expression]

value       ::= integer | real | string |
                "{" expression { , expression } "}" |
                "{}" | "USET" | "<" field{ , field} ">" |
                "[" expression { , expression} "]"

field       ::= var : expression
```

BOOK REVIEWS

PASCAL for the MACINTOSH, H. Ledgard and A. Singer., Addison-Wesley Publishing Company, 1985, ISBN 0 201 11772 X.

Reviewed by : C.A. Kelly,

This book gives a detailed introduction to PASCAL covering everything from the most basic data types to dynamic data structures and linked lists. It also covers all the basic control structures used in PASCAL including an introduction to top-down programming techniques.

All the graphics commands which are not defined in standard PASCAL, but are however included in Apple Macintosh PASCAL, are also described in detail. These special features are also used to great effect in many of the examples given.

Each section of this book is preceded by a dialogue between Sherloch Holmes (the teacher) and Watson (the student) where the solution of some kind of programming problem is attempted. The solution of the problem is given in the form of a PASCAL program and thereafter the technical detail for the solution and the relevant program constructs are discussed. Many programming exercises are also given at the end of most chapters which are extremely useful to most beginners in PASCAL.

For those in possesion of an Apple Macintosh personal computer and need an introduction to PASCAL this book is well worth purchasing. Futher anybody in the business of tutoring PASCAL could find the dialogues and the examples useful material. This book, however, is certainly not aimed at advanced users of PASCAL as it is only an introduction to the basic language constructs, and few advanced topics are covered.

A Beginner's Guide to Lisp, Tony Hasemer, Addison-Wesley, Wokingham, Berkshire, 1984. ISBN 0–201–14634–7. 257 pages.

Reviewed by: Philip Machanick, *University of the Witwatersrand, Johannesburg 2001*

In the wake of the discovery of Artificial Intelligence by the commercial world, there has been a flood of LISP books. Besides the usual criteria of readability, accuracy and completeness, a LISP book needs to pay particular attention to programming style. LISP does not embody a style—it is a basis for constructing your own. Approaches such as structured growth have developed through long experience. LISP is in most respects a functional language, but even functional programming (though becoming fashionable) is not universally embraced.

Scope

Hasemer aims at the small end (computers, not people) of the market. For this reason, he uses a minimal subset of LISP, with some features which are falling into disfavour—like dynamic scoping—presented without comment. Dialects of Common LISP (such as Golden Common LISP and ExperLISP) are appearing on cheaper machines: this book is in danger of becoming obsolete in this respect.

Aside from this, the book is ambitious for a beginner's text, going as far as building a simple production system by the last chapter. Although presentation of generalities before examples may confuse the uninitiated, applications are well-chosen to introduce AI (given the orientation to small machines). Topics include data structures, inference networks, ELIZA (the classic simple "natural language" system) and pattern matching, and searching trees.

Accuracy

Hasemer decided that, because there was no universally accepted standard, he would invent his own dialect. This approach was flawed when Donald Knuth introduced it in 1968 in his

still-incomplete *The Art of Computer Programming* series, and it has not improved with age. Not only does it ensure that *no one* is happy, but the author cannot test examples by running them (unless he writes his own interpreter). Many irritating errors result (the most embarassing: a program which is claimed to evaluate `(2 x 3) + (1 + 2)` as 3).

He also uses many odd variations on common knowledge and terminology. For instance, we are told that EXPRs save their arguments' "values away to be restored when the EXPR itself has been executed" (p. 19). This *can* be done (and is, in an interpreter presented later), but parameters are not usually passed like this. Other examples: nested function calls are described as "recursion"; a function called from another is called a "subroutine".

LISP Style

A strong point is that building tools (crucial with "toy" LISPs) is given major coverage. The language is presented as open-ended, ready for you to make your imprint. Beyond that—not so good. No doubt in the interests of efficiency (an idea of the intended audience is conveyed by the repeated mention of tapes as an alternative to disks), recursion is used relatively little. A functional style is not fully exploited. The major approach is PROGs with local variables (using SETQ to change ther values) and GO to make loops.

Approaches such as structured growth which are applicable to large-scale projects are absent. After all, this is a book for beginners. A bad point is the tendency towards allowing functions to grow too long (which is not so easy to do if you stick to a purely functional style). Unfortunately, this is a common flaw of programming texts (and programs in general).

Presentation

A weak point is the use of mechanistic discription—cons-cells are described in detail in Chapter 3. Many of the glitches are results of Hasemer's attempts at explaining *how* when *what* would have been good enough.

Otherwise, his conversational style may appeal to beginners as being non-threatening. Some will find it irritating. A worthwhile feature is the time spent on explaining how to *read* programs—a skill often neglected in programming courses. The balance could be moved more towards learning how to write: the examples are long, while many of the exercises are trivial.

Overall

This book will appeal most to those learning LISP for themselves, if they have a cheap interpreter with few tools and which is not close to a common dialect. The attention given to pointing out likely differences in implementations is particularly heplful. Academics are likely to be put off by the "non-standard" dialect, the errors and the programming style.

Books Received for Reviewing

Prentice-Hall International Series in Computer Science -

- Jim Welsh and Atholl Hay: **A model of Standard Pascal;** Prentice-Hall 1986, ISBN 0-13-586454-2; pp.483.
- C.A.R. Hoare and J.C. Shepherdson (Eds): **Mathematical Logic and Programming Languages;** Prentice-Hall 1985, ISBN 0-13-561465-1; pp. 184.

James S. Fritz, Charles F. Kaldenbach and Louis M. Progar: **Local Area Networks: Selection Guidelines;** Prentice-Hall 1985, ISBN 0-13-539552-6; pp. 106.

Alan Bonnet; **Artificial Intelligence - Promise and Performance** *translated by Jack Howlett;* Prentice-Hall 1985, ISBN 0-13-048869-0 (pbk); pp. 221.

John C. Molluzzo and Fred Buckley: **A First Course in Discrete Mathematics:** Wadsworth Publ. Co. 1986, ISBN 0-534-05310-6; pp. 507.

NOTES FOR CONTRIBUTORS

The purpose of the journal will be to publish original papers in any field of computing. Papers submitted may be research articles, review artilces and 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
INFOPLAN
Private Bag 3002
Monument Park 0105
South Africa

Form of manuscript

Manuscripts should be in double-space typing on one side only of sheets of A4 size with wide margins. Manuscripts produced using the Apple Macintosh will be welcomed. Authors should write concisely.

The first page should include the article title (which should be brief), the author's name and affiliation and address. Each paper must be accompanied by an abstract less than 200 words which will be printed at the beginning of the paper, together with an appropriate key word list and a list of relevant Computing Review categories.

Tables and figures

Tables and figures should not be included in the text, although tables and figures should be referred to in the printed text. Tables should be typed on separate sheets and should be numbered consecutively and titled.

Figures should also be supplied on separate sheets, and each should be clearly identified on the back in pencil and the authors name and figure number. Original line drawings (not photocopies) should be submitted and should include all the relevant details. Drawings etc., should be submitted and should include all relevant details. Photographs as illustrations should be avoided if possible. If this cannot be avoided, glossy bromide prints are required.

Symbols

Mathematical and other symbols may be either handwritten or typewritten. Greek letters and unusual symbols should be identified in the margin. Distinction should be made between capital and lower case letters; between the letter O and zero; between the letter I, the number one and prime; between K and kappa.

References

References should be listed at the end of the manuscript in alphabetic order of the author's name, and cited in the text in square brackets. Journal references should be arranged thus:

1. Ashcroft E. and Manna Z., The Translation of 'GOTO' Programs to 'WHILE' programs., *Proceedings of IFIP Congress 71*, North-Holland, Amsterdam, 250-255, 1972.
2. Bohm C. and Jacopini G., Flow Diagrams, Turing Machines and Languages with only Two Formation Rules., *Comm. ACM*, **9**, 366-371, 1966.
3. Ginsburg S., Mathematical Theory of Context-free Languages, McGraw Hill, NewYork, 1966.

Proofs and reprints

Proofs will be sent to the author to ensure that the papers have been correctly typeset and *not* for the addition of new material or major amendment to the texts. Excessive alterations may be disallowed. Corrected proofs must be returned to the production manager within three days to minimize the risk of the author's contribution having to be held over to a later issue.

Only orginal 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.

Book reviews

The journal has a section on book reviews and is keen to consider any contributions for this section.