Relational Pascal data base interface

S. Alagić and A. Kulenović

Faculty of Electrical Engineering, University of Sarajevo, Topličká bb, 71113 Sarajevo, Yugoslavia

The modification of Pascal proposed in the paper is based on the relation type which combines the properties of the set type and the file type of Pascal and includes properly these two Pascal types. The relation type is equipped with both high level (set oriented) relational operations and low level primitives for operating on particular tuples of relations. Within the relational framework a novel and unified treatment of images and base relations is given. This approach permits isolation of those issues in optimisation and decomposition which can be treated independently of the details of the procedural decomposition, clarifies and simplifies the decomposition problem. The proposed programming language is smaller and simpler than Pascal and still very powerful in handling relational data bases. Examples of nonprocedural and procedural decomposition are presented in order to demonstrate the suitability of the proposed features and their implications on the architecture of the supporting system.

(Received May 1979; revised February 1980)

1. The relation type

The form of the denotation of this type is

relation of ⟨base type⟩ (1.1)

where the base type may be a simple type, an array of characters (string) or a record type, whose fields have types which are either simple or arrays of characters (strings). The value of a variable of a relation type is a set of tuples. Since relations are sets, the properties of relations are in many ways similar to the properties of Pascal sets. The essential differences are:

1. The base type of a relation may be a record type or an array of characters.
2. The size of relations is enormous in comparison with the size of Pascal sets, and thus their mode of implementation involves secondary storage.
3. In addition to the usual way of constructing sets in Pascal, a relation constructor of a general form is available for relations. This constructor includes, as particular cases, the operations of projection, restriction and join of the relational algebra.
4. A special form of statement (**foreach**) is available for relations; its form parallels the form of the relation constructor.

Base relations appear in a Pascal user program as external objects (just like external files). They must be defined in the program heading. Those fields in the base relations irrelevant to the user program may be omitted, otherwise the names of relations, field identifiers and their types must conform to those of base relations.

As an example, we give declarations of relations describing the data base of a department store. This example will be used in subsequent sections.

```
type string = array [1 . . 20] of char;
     deptype = (toy, shoe, furniture, appliances, food,
                men, ladies, cosmetics, admin);
     jobtype = (teller, accountant, assistant, manager);
     emprec = record name: string;
                     dept: deptype;
                     mgr: string;
                     sal: integer;
                     job: jobtype
              end;
     supplyrec = record supplier: string;
                        item: integer;
                        vol: integer
                 end;
     salesrec = record dept: deptype;
                       item: integer;
                       vol: integer
                end;
     locrec = record dept: deptype;
                     floor: 1 . . 20
              end;
var  emp: relation of emprec;
     sales: relation of salesrec;
     supply: relation of supplyrec;
     loc: relation of locrec;                          (1.2)
```

2. Relation constructor

Relational expressions are similar to set expressions in Pascal. The empty relation is denoted as [ ]. In order to denote a relation which consists of elements $r_1, r_2, \ldots, r_n$, we write $[r_1, r_2, \ldots, r_n]$. $r_1, r_2, \ldots, r_n$ are expressions which, when evaluated, yield the tuples of the constructed relation. The most general form of the relation constructor is

```
[each   ⟨list of expressions⟩
 for    ⟨list of control variables⟩
 in     ⟨list of relation variables⟩
 where  ⟨qualification expression⟩].                   (2.1)
```

The control variables correspond to the relation variables in the list which follows **in** and this left to right correspondence determines their type (range). The expressions in the list after **each** determine how fields of the tuples of the constructed relation are computed (selected). The length of this list determines the degree of the constructed relation. The boolean expression after **where** is the restriction condition. Only tuples satisfying this condition qualify for the selection operation. This construct combines the operations of projection, restriction and join of the relational algebra.

In the example below a new relation *underpaid* is constructed consisting of names and salaries of assistants who earn less than *10000*.

```
var underpaid: relation of record name: string;
                                  sal: integer
                           end;
begin underpaid := [each x.name, x.sal
                    for x in emp
                    where (x.job = assistant)
                    and (x.sal < 10000)]
end                                                    (2.2)
```

In order to construct the relation whose tuples consist of names, salaries and department names of assistants located on the first

CCC-0010-4620/81/0024-0112 $03.00

floor, we write the expression:

```
[each x.name, x.sal, x.dept
 for  x, y in emp, loc
 where (x.job = assistant)
 and   (x.dept = y.dept)
 and   (y.floor = 1)] ,                    (2.3)
```

which, in addition to the operations of projection and restriction includes also the operation of joining the relation *emp* and *loc* (*x.dept* = *y.dept*) over the field *dept*.

3. Set operators

All the set operators from Pascal are applicable to relations. These operators are membership (**in**), subset operations (<, <=, >=, >), equality and nonequality of sets (=, <>), union (+), intersection (*) and difference (−).

In this section we give examples illustrating how some of these operators may be used. In order to construct the relation of all items sold by departments on the second floor we may write:

```
[each  x.item
 for   x in sales
 where x.dept in [each  y.dept
                  for   y in loc
                  where y.floor = 2]] .    (3.1)
```

An alternative way, based on the explicit use of the join operation, would be:

```
[each  x.item
 for   x, y in sales, loc
 where (x.dept = y.dept)
 and   (y.floor = 2)] .                    (3.2)
```

The use of the set union is illustrated by the following example, in which a new tuple is inserted into the relation *emp*:

```
var newemp: emprec;
begin with newemp do
    begin name := 'anderson';
          dept := toy;
          job := assistant;
          sal := 7000;
          mgr := 'jones'
    end;
    emp := emp + [newemp]
end .                                      (3.3)
```

The final example in this section illustrates how the operation of division of the relational algebra is expressed within this framework. The value of the expression (3.4) is the set of names of those suppliers who supply all items sold in the cosmetics department:

```
[each  x.supplier
 for   x in supply
 where [each y.item
        for y in supply
        where y.supplier = x.supplier]
        >=
       [each  x.item
        for   x in sales
        where x.dept = cosmetics]].        (3.4)
```

We use the example (3.4) to point out the advantages of block structure which avoids the problems of block label notation discussed in Astrahan *et al.* (1976) and Chamberlin *et al.* (1976). In the above example we have three scopes, each of which coincides with one relation constructor. The outer scope has a local variable *x*. Two inner scopes, nested within the outer scope, contain local variables *y* (first) and *x* (second). Nesting of relation constructors amounts to nesting of procedures into which these constructors are decomposed in the manner to be described in later sections. This means that the block structure just described is expressed in terms of the block structure of Pascal.

4. Standard functions

The following standard functions over relations are available: *card*, which gives the cardinality of a relation, *sum*, *max*, *min* and *avg* which give the sum, the maximum, the minimum and the average value respectively of the relation to which they are applied. The functions *sum*, *max*, *min* and *avg* may be applied only to relations of degree one whose tuples are either of type *real*, *integer* or a subrange of *integer*.

The following relational expression constructs the relation of those employees whose salary is greater than that of any employee in the shoe department:

```
[each  x.name
 for   x in emp
 where x.sal > max ([each  x.sal
                     for   x in emp
                     where x.dept = shoe])].   (4.1)
```

5. Foreach statement

foreach statement is used for specifying actions upon relations Its form is chosen in such a way that it parallels the relation constructor:

```
foreach ⟨list of control variables⟩
in      ⟨list of relation variables⟩
where   ⟨qualification expression⟩
do      ⟨statement⟩ .                      (5.1)
```

Control variables in the list after **foreach** correspond, in the left-to-right order, to the relation variables in the list after **in**, and their types are the respective base types of these relations. The scope of the control variables is the statement following **do.** This statement specifies the action to be performed upon the selected tuples. This action may change the values of control variables. Only those tuples satisfying the qualification condition (a boolean expression) are subject to this action. We shall first show how to use this statement in order to perform updating of selected tuples in a relation.

Give a 10 percent raise to Adams if he works on the first floor:

```
foreach  x, y in emp, loc
where    (x.name = 'adams')
 and     (x.dept = y.dept)
 and     (y.floor = 1)
do       x.sal := x.sal * 1.1 .            (5.2)
```

foreach statement is also used to delete selected tuples from a relation. For example, in order to delete all employees who work for departments on the first floor we may write:

```
foreach  x, y in emp, loc
where    (x.dept = y.dept)
 and     (y.floor = 1)
do       emp := emp − [x] .                (5.3)
```

It may also be used in a similar fashion to insert selected tuples into a relation.

6. Images and high level decomposition

The features of extended Pascal described in the previous sections are intended for (far) end users of the system. In the subsequent sections we describe features whose use must be much more restrictive, since they offer much more explicit control over the base relations. In this section we focus to one such feature, an image, and its consequences.

An image is a relation which determines logical ordering of tuples of some other relation with respect to values in one or more (sort) fields of the relation. The base type of an image

consists of these sort fields and a pointer field. For a given value of the sort fields, the value of the pointer field determines the tuple of the base relation with the given value of the sort fields. For example, suppose that we want to define an ordering of the relation *emp* on the sort field name. An image which determines this ordering would be defined as follows:

```
var namemp: relation of record name: string:
                                ref: ↑emprec
                end;                                    (6.1)
```

and it is created by a call of the procedure *createimage* in which the parameters specifying the identifier of the image and of the relation over which it is defined must be supplied. The pointer type is here the pointer type of Pascal adapted for the specific purposes of secondary storage. It is as in Pascal, a linearly ordered set of values (see Hoare and Wirth, 1973; Jensen and Wirth, 1975) which point to values of the type to which the pointer type is bound. The use of pointer fields in images is subject to a very restrictive discipline which does not allow the users to change them (they appear as read-only variables). All the changes are performed by the system.

One or more images may be created over a relation. Among them, at most one image may have the clustering property. A clustering image is an image in which the physical placement of tuples of the underlying relation corresponds to the logical ordering determined by the image. If the values of the sort fields of an image constitute a candidate key of the underlying relation, the image is called unique.

The user who is aware of the existence of images over a relation is in a position to specify much more efficient actions upon the relation. This follows from the fact that an image is a much smaller relation than the base relation over which it is defined, and thus searching an image is much faster than searching a relation. This simple argument based on cardinalities has, of course, much more involved implementation aspects, of which the user need not be aware at all (images are implemented using dynamic multilevel index structures). Furthermore, not only the presence of images, but also their clustering property, greatly affects the efficiency of various actions.

While the knowledgeable user may specify its actions in a more specific form using images, the end user who is not aware of them will use the features described in the previous sections, and his programs are subject to a decomposition procedure performed by the system in which the existence of images and their properties are explored and exploited to obtain a more specific and more efficient program specification. This procedure is called high level decomposition since its result is still a program expressed in terms of high level, set (relation) oriented primitives, as opposed to a subsequent decomposition step whose result is a procedural form of the program expressed in terms of tuple-at-a-time primitives. The high level decomposition procedure will be illustrated by two examples.

Consider the statement:

```
result := [each  x.name, y.floor
           for   x, y in emp, loc
           where x.dept = y.dept]                       (6.2)
```

and suppose that there exist images *empdept* and *locdept* over the relations *emp* and *loc* respectively. The sort field of these images is the joining field *dept*. Given these assumptions, there are two decompositions of the statement (6.2). One of them is presented in (6.3), and in the other the roles of *empdept* and *locdept* are interchanged.

```
type namefloor = record name: string;
                        floor: 1 . . 20
                 end;
var result: relation of namefloor;
    buff: namefloor;
    empdept: relation of record dept: deptype;
                                ref: ↑emprec
                        end;
    locdept: relation of record dept: deptype;
                                ref: ↑locrec
                        end;
begin result := [ ];
      foreach x in empdept
      do foreach y in locdept do
         begin buff.name := x.ref↑.name;
               buff.floor := y.ref↑.floor;
               result := result + [buff]
         end
end .                                                   (6.3)
```

The difference between the two decompositions is thus which image controls the outer loop. In (6.3) it is *empdept*. The optimiser must choose one of the two decompositions. (6.3) is a clear choice if *empdept* is not a clustering image and *empdept* is. In other cases (either both images are clustering or both are not clustering) the optimiser must evaluate performance formulas in order to make a choice.

The next example is more involved. Consider the statement:

```
result := [each x.name, y.floor
           for   x, y in emp, loc
           where (x.job = assistant)
             and (y.floor = 4)
             and (x.dept = y.dept)]                     (6.4)
```

and suppose that there exist nonclustering images *empjob* and *empdept* over the relation *emp* and *locfloor* and *locdept* over the relation *loc*. The sort fields of these images are specified in the following declarations:

```
var empjob: relation of record job: jobtype;
                               ref: ↑emprec
                       end;
    locfloor: relation of record floor: 1 . . 20;
                               ref: ↑locrec
                       end;
    empdept: relation of record dept: deptype;
                               ref: ↑emprec
                       end;
    locdept: relation of record dept: deptype;
                               ref: ↑locrec
                       end;                             (6.5)
```

Under these assumptions, a possible decomposition of (6.4) which is based on the TID algorithm of Astrahan *et al.* (1976) is presented in (6.6). Whether this decomposition will be chosen is decided by the optimiser evaluating performance formulae for this and other possible decompositions of (6.4).

```
var r1: relation of ↑emprec;
    r2: relation of ↑locrec;
    r3: relation of record ref1: ↑emprec;
                           ref2: ↑locrec
                   end;
        buff: record name: string;
                     floor: 1 . . 20
              end;
begin   r1 := [each  x.ref
               for   x in empjob
               where x.job = assistant];
        r2 := [each  x.ref
               for   x in locfloor
               where x.floor = 4];
        r3 := [each  x.ref, y.ref
               for   x, y in empdept, locdept
               where x.dept = y.dept];
```

```
result := [ ];
foreach x in r3
where (x.ref1 in r1)
 and (x.ref2 in r2)
do begin buff.name := x.ref1↑.name;
         buff.floor := x.ref2↑.floor;
         result := result + [buff]
   end
end .                                                    (6.6)
```

7. Low level interface

The low level interface contains tuple-at-a-time primitives upon relations, which are similar to the file primitives of Pascal. In fact, the relation type, as it appears at this level, includes properly the file type of Pascal.

In view of the fact that the low level interface is powerful in terms of what it enables the user to do with the base relations and images over them, granting these facilities must be carefully controlled.

The low level primitives upon relations and images are summarised briefly and informally below. Examples of their use will be given in the next section. In what follows f denotes a relation which may be an image unless otherwise specified.

rewrite (f) The empty relation is assigned to f.

reset (f) Every relation has a cursor associated with it. The cursor determines the current position within the relation and thus the current tuple of the relation, which equals the value of the variable $f\uparrow$. *reset* (f) sets the cursor to the first tuple of the relation f. If f is an image, the first tuple is determined according to the sort fields of the image.

get (f) Applicable only if the cursor of the relation f does not point to the end of f (**not** *eof* (f) must hold) . *get* (f) moves the cursor one position in the forward direction. If f is an image, the forward direction is determined by the sort fields of the image.

get (f, k) k is a variable whose type is the base type of f . *get* (f, k) positions the cursor of f to the first tuple of f which equals k. If no such tuple exist in f, *eof* (f) is set to true. If f is an image, setting the pointer field in k before calling *get* (f, k) has no effect.

put (f) Applicable only if the cursor of f points to the end of f (*eof* (f) must hold) . *put* (f) appends $f\uparrow$ to f. All images over f are updated . *eof* (f) remains true. If f is an image possible setting of the pointer field in $f\uparrow$ has no effect. This field will be set in the appended tuple by the system.

put (f, t) t is a variable of the base type of f. If f is an image, possible setting of the pointer field of $f\uparrow$ has no effect . *put* (f, t) inserts t into f according to the implementator defined algorithm. If f is an image, this insertion preserves the linear ordering of the image. If f is a base relation, all images over f are updated to reflect the insertion of t.

delete (p) The object identified by p is deleted. If p is a relation then p is deleted. If p is of the form $f\uparrow$ then the current tuple of the relation f ($f\uparrow$) is deleted. If p is a pointer, then the tuple pointed to by p is deleted. In the last two cases all the necessary updating of images is performed.

eof (f) This expression evaluates to true if the cursor of f points to the end of f.

eod (f) Evaluates to true when the end of a sequence of tuples with the same value of the sort fields (duplicates) is reached.

resetd (f) resets the cursor of f to the beginning of the current sequence of duplicates.

8. Low level decomposition

In this section we give examples of decomposition of high level relational constructs into procedural form expressed in terms of low level primitives. The purpose of the chosen examples is to demonstrate the suitability of the low level interface.

Consider first the statement:

```
result := [each x.name, x.job
           for  x in emp]                                (8.1)
```

where result is a relation variable of the type **relation of record** *name*: *string*; *job*: *jobtype* **end** . Suppose that there is a clustering image over *emp* whose sort field is one of the selected fields in the above query. Let that field be *name*. Then in order to compute the value of *result*, the relation *emp* is processed according to its clustering image. During this processing only fields *name* and *job* are selected. Tuples constructed in such a way are appended to the relation *result* in the order of supply.

For the above assumptions when removal of duplicates is not necessary we obtain the following decomposition of (8.1):

```
var result: relation of record name: string; job: jobtype end;
    nameimage: relation of record name: string;
                                   ref: ↑emprec end;
begin rewrite (result);
      reset (nameimage);
      while not eof (nameimage) do
      begin result↑.name := nameimage↑.name;
            result↑.job := nameimage↑.ref↑.job;
            put (result); get (nameimage)
      end
end .                                                    (8.2)
```

The previous example shows how the operation of projection is decomposed into a sequence of low level primitives. In the following example we show how a relation constructor which includes both projection and restriction, is decomposed.

```
result := [each  x.name, x.dept
           for   x in emp
           where (x.job = assistant)
            and  (x.sal < 10000)] .                      (8.3)
```

Suppose that there is an image over the relation *emp* with the sort field *job*. Then in the procedure which evaluates *result*, given in (8.4), the collection of tuples with the field *job* equal to *assistant* are accessed, and those among them which satisfy the condition *sal* < 10000 are selected. If the removal of duplicates is necessary (in (8.4) we assumed it is not), it is performed in the process of selecting the fields *name* and *dept*. The procedure (8.4) is given for the general case when the image on *job* is not unique.

```
type element = record job: jobtype;
                      ref: ↑emprec
               end;
var result: relation of record name: string;
                               dept: deptype
                    end;
    jobimage: relation of element;
    argument: element;
begin rewrite (result);
      argument. job := assistant;
      get (jobimage, argument);
      if not eof (jobimage) then
      repeat if jobimage↑.ref↑.sal < 10,000 then
             begin result↑.name := jobimage↑.ref↑.name;
                   result↑.dept := jobimage↑.ref↑.dept;
                   put (result)
             end;
             get (jobimage)
      until eod (jobimage)
end .                                                    (8.4)
```

As the final example we present a procedural decomposition of (6.3) which was obtained as the result of high level decomposition. At the procedural level (8.5) amounts to tuplewise sequential reading of the images *empdept* and *locdept*. For each matching pair of sort fields of these images, the fields *name* and *floor* are selected from the corresponding tuples in the relations *emp* and *loc* respectively. In (8.5) we assumed that removal of duplicates is not necessary.

```
var empdept: relation of record dept: deptype;
                                ref: ↑emprec
                           end;
    locdept: relation of record dept: deptype;
                                ref: ↑locrec
                           end;
    result:  relation of record name: string;
                                floor: 1 . . 20
                           end;
begin reset (empdet); reset (locdept); rewrite (result);
      while not (eof (empdept) or eof (locdept)) do
      begin if empdept↑.dept < locdept↑.dept then
                get (empdept)
            else if empdept↑.dept > locdept↑.dept then
                     get (locdept)
            else begin repeat resetd (locdept);
                              repeat result↑.name :=
                                       empdept↑.ref↑. name;
                                     result↑. floor :=
                                       locdept↑.ref↑. floor;
                                     get (locdept)
                              until eod (locdept);
                              get (empdept)
                         until eod (empdept);
                         get (empdept); get (locdept)
                 end
      end
end .                                                    (8.5)
```

Conclusions

Starting from a subset of Pascal which is essentially Pascal-S (Wirth, 1975) we propose its extension to a complete programming language based on the concepts and notation of Pascal. This extension includes a relational data sublanguage. Our aim was to obtain a simple and powerful programming language based on Pascal which would be used in the area of relational data bases.

The features which extend Pascal-S are structured into a hierarchy of three levels which are reflected in the architecture of the supporting system. Since the level of control over the data base varies from level to level, granting features of lower levels must be carefully controlled.

The highest level is purely set oriented and relational. The relational type which extends Pascal-S appears at this level as a modified set type of Pascal. Apart from the standard set operations of the set type, the operations of projection, restriction and join of the relational algebra are included as special cases of the relation constructor. **foreach** statement is added for concise specification of actions upon relations. In comparison with Schmidt's proposal (1977), we made some significant changes of which we mention two: (*a*) Quantifiers are excluded. The justification for this is the fact that the data sublanguage is still relationally complete, as pointed out by Chamberlin and Boyce (1974), and much simpler, (*b*) **foreach** statement is generalised in that it may include a list of control variables as in the relation constructor. This natural generalisation makes the two constructs completely parallel (one is an expression, the other is a statement). These two changes taken together simplify considerably the sublanguage and make it more convenient and expressive.

The second level, although still completely relational and set oriented, allows specification of orderings of relations which are supported by access paths. These orderings are, following Astrahan *et al.* (1976), called images. The novelty is a unified treatment of images and base relations as objects of type relation, and introduction of a modified (more restrictive) pointer type of Pascal. We showed using examples that our relational treatment of images allows isolation of those issues in optimisation and decomposition which can be treated independently of the details of the procedural decomposition. In the process of this high level decomposition, the existence of images upon relations is exploited to obtain a modified but still set oriented program specification, which is now reduced to a form whose translation into the procedural form is fairly straightforward. Choosing one among several possible strategies for high level decomposition is based on the performance formulae to be presented elsewhere.

The lowest level consists of tuple-at-a-time primitives upon relations which extend the file primitives of Pascal. At this level, the relation type together with low level primitives includes the file type of Pascal as a special case. This rounds up our design in which all the types of Pascal are present with suitable modifications where necessary.

Although we borrow many ideas from Chamberlin and Boyce (1974) and Chamberlin *et al.* (1976), most significant differences come from the properties of Pascal, its discipline of types and very careful design of the language in general. The result of adopting strictly the philosophy of Pascal in extending Pascal-S with various features is much more precise and consistent language than Chamberlin *et al.* (1976), which can be axiomatised in the style of the axiomatic definition of Pascal (Hoare and Wirth, 1973). This holds even for the lowest level, and it would be very difficult to achieve in Chamberlin *et al* (1976). Some of the advantages mentioned in the paper come from the block structure (nesting of relation constructors, precise use of standard functions and relations, unified treatment of relations and images as sets of tuples, etc.).

Acknowledgement

Research presented in the paper was supported by Republička zajednica za naučni rad SR Bosne i Hercegovine.

References

ALAGIĆ, S. and ARBIB, M. A. (1978). *The Design of Well-Structured and Correct Programs*, Springer-Verlag.

ALAGIĆ, S., JOVIĆ, R. and RIDJANOVIĆ, DŽ. (1977). A hierarchical host language system based on B-trees, *Proceedings of IVth Annual Congress AICA*, Pisa.

ASTRAHAN, M. M. *et al.* (1976). System-R: Relational approach to database management, *ACM Transactions on Database Systems*, Vol. 1 No. 2.

CHAMBERLIN, D. D. and BOYCE, R. F. (1974). Sequel: A structured English query language, *ACM-SIGFIDET Workshop on Data Description, Access and Control.*

CHAMBERLIN, D. D. *et al.* (1976). Sequel 2: A unified approach to data definition, access and control, *IBM Journal of Research and Development*, Vol. 20 No. 6.

CODD, E. F. (1972). Relational Completeness of Data Base Sublanguages, in *Data Base Systems*, Courant Computer Sci Symp 6th, Prentice-Hall.

HOARE, C. A. R. and WIRTH, N. (1973). An axiomatic definition of the programming language Pascal, *Acta Informatica*, Vol. 2.

JENSEN, K. and WIRTH, N. (1975). *Pascal User Manual and Report*, Springer-Verlag.
SCHMIDT, J. W. (1977). Some high-level language constructs for data of type relation, *ACM Transactions on Database Systems*, Vol. 2 No. 3.
SMITH, J. M. and CHANG, P. Y. T. (1975). Optimizing the performance of a relational algebra database interface, *CACM*, Vol. 18 No. 10.
STONEBRAKER, M., WONG, E., KREPS, P. and HELD, G. (1976). The design and implementation of INGRES, *ACM Transactions of Database Systems*, Vol. 1 No. 3.
WEDEKIND, H. (1974). On the selection of access paths in database systems, in *Data Base Management* North-Holland.
WIRTH N. (1975). *Pascal-S: A subset and its implementation*, ETH, Zurich.

Book reviews

Software Psychology—Human Factors in Computer and Information Systems, by Ben Schneiderman, 1980; 320 pages. (*Prentice-Hall*, £16·20)

As the range of subject matter covered may not be entirely clear from the title, I will start with a list of chapter headings. These are: 1. Motivation for a psychological approach; 2. Research methods; 3. Programming as human performance; 4. Programming style; 5. Software quality evaluation; 6. Team organisations and group processes; 7. Database systems and data models; 8. Database query and manipulation languages; 9. Natural language; 10. Interactive interface issues; 11. Designing interactive systems; 12. Computer power to, of and by the people.

I found the book both readable and interesting. It is mostly devoted to discussions and comparisons of different approaches and methods in the areas covered by the chapter headings, supported by extensive references to published work. The book has a 20 page bibliography. Two examples of topics covered in some detail are the value of flowcharting in program development and comprehension, and different approaches to software quality evaluation. A satisfactory feature is that, where appropriate, the statistical significance of results is quoted and the author usually attempts to reach a view on what the (sometimes conflicting) evidence means. Although—perhaps inevitably—firm conclusions are rare, the summaries of research evidence on a range of software topics could be of interest to quite a wide spectrum of readers. At the end of each chapter is a useful 'practitoner's summary' followed by a 'researcher's agenda' indicating where further work is needed.

There are, however, some gaps, and it is not entirely clear how the main areas covered in the book have been selected. For example, the direct interaction between human beings and computers via various types of language is given extensive treatment. On the other hand the problem of establishing, defining and specifying requirements for software, which is at least as difficult and which has considerable psychological content, is hardly covered at all. Perhaps this is because there has been little psychological research in this area due to a number of factors but one would have expected it to be discussed and listed as an important field for future work.

Software Psychology is a valuable book for anyone seriously concerned with the practice of programming and particularly those involved in research into human factors aspects. Those with a more general interest in computing and software will find much that is useful, although for such readers the book, at £16·20, will probably be one to borrow rather than buy. It is well produced with clear printing and I did not spot one typographical error.

J. N. G. BRITTAN (Chertsey)

Programming Standard Pascal, by R. C. Holt and J. N. P. Hume, 1980; 381 pages. (*Prentice-Hall* £7·75)

This book, like most elementary programming texts, is good in parts. Its aim is to teach basic programming in Pascal by using series of subsets of the language that include one another like a set of Russian dolls. The technique was developed for PL/I by R. Holt and D. Wortman. (The Venn diagram used to illustrate the subset idea is labelled wrongly, since the largest, all enclosing, set is labelled PS/1, the name of the smallest subset that the authors define.)

The first two chapters are devoted to an introduction to computers and programming and a justification of the subset approach. This seems overlong; especially when trying to explain structured programming to readers who cannot program. Each of the following three chapters introduces a new subset of the language. The first contains only arithmetic expressions and printing of values; it seems rather small, and could profitably have been combined with the second subset which introduces variables and declaratives as well. The third of these subsets is far too large—entitled 'Control flow'. The chapter includes **while, repeat** and **for** loops, **if** and **case** statements, **Boolean** variables and expressions—all within 20 pages. I would introduce this wide selection of topics much more slowly—probably leaving the **case** statement until enumerated types had been introduced and develop the **for** statement with **arrays.**

After the introduction of **arrays** and the various sorts of type (subrange, named, enumerated), the presentation improves a great deal. A chapter is devoted to top-down development of solutions and choice of data structures. Procedures and functions are introduced competently.

An excellent section now follows with chapters devoted to Modular programming, Searching + sorting, Making sure the program works and Data structures. Records and pointers are painlessly introduced here. The chapter on data structures introduces stacks, queues and trees. Unfortunately, recursive procedures are introduced, and explained in terms of using a stack to print a list in reverse order. This is a most un-natural use for them—even worse than the usual function to calculate $n!$. The example implementations given for stacks and queues using arrays contain no check on overflow or underflow. This is most serious in the case of the queue implementation since modular arithmetic is used to keep the indices within the array bounds.

The remaining chapters of the book are again of limited usefulness. Scientific calculations gives a procedure for plotting approximate curves on the line printer, but is otherwise little more than a superficial overview of the problems attacked by numerical analysis. Similarly the chapter Numerical methods only presents an arm's length picture of the problems involved.

A short chapter gives examples of programs in other languages, and the two final chapters are devoted to a discussion of machine language and assemblers and the development of a compiler for a (very) restricted subset of Pascal.

Each chapter concludes with a short summary and a set of exercises. There were few typographical errors and the only factual error I noted was the statement 'In division the relative error of the quotient is the difference between the relative errors of the divisor and the dividend'. (It is, in fact, the sum).

P. KING (Newcastle)

Architecture and the Microprocessor, by John Paterson, 1980; 229 pages. (*John Wiley*, £13·80)

Despite its title this book is not concerned too much with architecture or the microprocessor—and most certainly not about microprocessor architecture. It is in fact a middlebrow Sunday supplement social history which looks at how cheap computing power might affect the practice of architecture. To do this the author traces the development of the art of architecture showing how changes in society have affected the architect and goes on to discuss how current changes in our society, and the microprocessor revolution in particular, might shape our future environment. The book may be an interesting diversion for the computer professional in two ways. Firstly, it is a good example of the impact of computers on society generally. Secondly, because the design of buildings is, in many ways, analogous to the design of computer software, this book may give an insight into the type of problem which may soon be confronting computer scientists.

ALAN H. BRIDGES (Glasgow)