SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 9, 671–683 (1979)

Strings and the Sequence Abstraction in Pascal

A. H. J. SALE
Department of Information Science, University of Tasmania, Tasmania

SUMMARY

This paper examines the sequence abstraction known in Pascal as the 'file', and shows how sequences of characters ('strings' in the SNOBOL sense) may be cleanly fitted into Pascal-like languages. The specific problems of providing the suggested facilities as an experimental extension to Pascal are examined.

KEY WORDS Pascal Strings Sequence abstraction

> 'Both men and ships live in an unstable element, are subject to subtle and powerful influences and want to have their merits understood, rather than their faults found out.
>
> 'It is not what your ship will *not* do that you want to know to get on terms of successful partnership with her; it is, rather, that you ought to have a precise knowledge of what she will do for you when called upon to put forth what is in her by a sympathetic touch.'
>
> JOSEPH CONRAD, *The Mirror of the Sea*

1. INTRODUCTION

As is well known, the programming language Pascal[1] is designed to give programmers access to data-types which are based on good abstractions. One of these is the packed array of characters, whose size is compile-time determined. This type is usually thought of as the only one available to Pascal programmers for handling sequences of characters as in command recognizers and other applications, and while it is adequate for these purposes, there have been some unfortunate consequences of this attitude. One of these is the regrettable lack of quality in many Pascal implementations which limit the significant characters in identifiers to the first eight, ten or twelve characters.

Whatever the cause for this state of affairs, there has been some pressure from implementors and users of Pascal for facilities for handling strings of characters of variable length. No doubt several such schemes have been implemented; one such was undertaken by Professor Ken Bowles' group at the University of California at San Diego, and another at the University of Tasmania.

This paper was stimulated by contact with the UCSD implementation, and by the realization by the author that variable-length character strings could be put on a firm axiomatic and abstract basis within the Pascal framework. The paper displays the end-result of this realization.

0038–0644/79/0809–0671 $01.00

Received 9 October 1978

2. TERMINOLOGY

In Pascal, there is a data structuring method which appears as

file of *thing*

This is singularly poorly named, given the plethora of meanings attached to the word *file* in computing circles. While I shall continue to use the Pascal terminology in the paper, readers should note that the underlying abstraction is more clearly expressed as a

sequence of *thing*

The abstraction consists of a variable number of objects of identical type, arranged in a sequence. Disks, end-of-line markers and other features are not relevant to the underlying abstraction,[2] though they may be relevant to some implementation details.

In addition, the Revised Report defines the lexical token

⟨*string*⟩ ::= '⟨*character*⟩ {⟨*character*⟩}'

and goes on to specify that this token may be a constant of the type *char*, or of the types **packed array**[1...*n*] **of** *char* (where $n > 1$). Since this is likely to give rise to endless confusion, I shall refer to these tokens only as *string constants*. The term *string* will be reserved for a type which denotes a sequence of characters whose length varies during execution, in a similar manner to the corresponding concept in SNOBOL.

3. THE BASIC ABSTRACTION

No more than a few moments of thought suffices to convince anyone, once presented with the thought, that an appropriate abstraction for a string is a **file of** *char*. It is self-evidently obvious, once the distracting notions have been disposed of.[2]

The indications are simple: the **file** is the only Pascal structuring method which allows for variable-size contents, apart from variant records (which require explicit enumeration of the possible variations). The only feasible alternative is to consider a string as a varying-length **array of** *char*; this can readily be shown to be a direct subversion of the basic design principles of Pascal, and to lead to many ramifications in the implementation.

The basic abstraction of a **file** is that of a type of potentially infinite cardinality, composed of objects of identical type arranged in a linear sequence, and with several useful operations defined upon the abstraction. Amongst these are those of reading through the sequence, of building up a sequence by concatenating elements together, of concatenating existing sequences, of examining the properties of the sequence and so on.

Within Pascal, however, the basic abstraction has been added to, and hedged in with restrictions, so as to make the abstraction capable of efficiently matching real physical world computer files.[3] Hence the name. These accretions to the basic concept include the introduction of a file buffer variable (having evolved from the conceptually simpler file window), the get and put operations, and a series of restrictions on the use of file structuring. In addition, the pre-declared type *text* is a funny sort of file of char which is line-organized according to the following EBNF syntax

{{*character*} *line-marker*} *end-of-file*

In many ways, this pre-declared type might have been better thought of as being

type
text = **packed file of packed file of** *extendedchar*;

where the type *extendedchar* includes the line-marker (but type *char* does not).

However, the important point to realize is that the file, as embedded in Pascal, has acquired some of these barnacles. They are not an essential part of the sequence abstraction, and I shall have to slough some of them off to achieve my purpose of exposing another use of the abstraction.

4. THE STRING TYPE

I propose that there may be a new Pascal pre-declared type in some implementations

```
type
  string = file of char;
```

which has some properties which are not common to other file types.

The first additional property is to remove the implementation restriction on the assignment of files. I therefore allow

```
var
  S1, S2 : string;
begin
  ...
  S1 := S2;
```

The semantics of assignment are straightforward: the entire file $S2$ is copied to $S1$, replacing any previous value of $S1$. State information associated with either file (read/write status, file window positioning) is not defined after the assignment. Note that this is the only sensible way to treat assignment with its underlying abstraction. (The few Pascal implementations which allow general file assignment but interpret it simply as a copying of the file descriptor clearly do not recognize the underlying abstraction.)

Given that file-assignment is well defined, it then makes sense to permit files of type string to be passed as value parameters (which involves an implicit assignment) as well as the variable (**var**) parameters now permitted. It is also sensible to relax the common implementation restrictions on incorporating and manipulating objects of type string within other structuring methods, and also on creating objects containing strings via the *new* procedure. Examples of legal constructs:

```
var
  SR, SR1 : record
              processed : boolean;
              command : string;
            end;
       SA : array[weekdaytype] of string;
       SP : ↑string;
begin
  ...
  new(SP);
  SP↑ := SA[Tuesday];
  SR := SR1;
```

The second major restriction to relax is that forbidding comparison of structured types. The operators $<$, $\leqslant$, $=$, $\geqslant$, $>$ and $\neq$ are defined for comparisons between objects of string type. The semantics are more complex than for assignment:

(a) Two strings are equal only if their lengths are equal, and the char components are one-for-one equal.
(b) The order of two unequal strings is determined by the ordering of the first item in which they differ; or if there is no such item, the shorter string precedes the longer string which it begins.

All other comparison operations can be framed in terms of these two rules; in every case the associated file states are left undefined (as for assignment).

It is important to realize that the semantics of assignment and comparison include the empty string. This is not an erroneous value, nor yet an undefined value or an implementation-dependent one, but a proper value for an object of string type.

One final change is needed to complete the essential components of the string type properties: the ability to have constants of the type. Fortunately, much of the necessary mechanism already exists in embryo form. Section 4 of the Revised Report says:

> 'Sequences of characters enclosed by quote marks are called strings. Strings consisting of a single character are the constants of the standard type char (see 6.1.2). Strings consisting of n (>1) enclosed characters are the constants of the types (see 6.2.1)
> **packed array** [1..*n*] **of** char
> Note: If the string is to contain a quote mark, then this quote mark is to be written twice.
> ⟨string⟩ ::= '⟨character⟩ {⟨character⟩}'

Clearly, the object described is really a sequence, and its pressed use for these types is less natural than as a sequence. Consequently, if I substitute 'string constant' for 'string', allow empty string constants, and rewrite the definition, then the necessary constants of type string fit naturally into the Pascal framework. The relevant section of a revised Report should then read:

> 'Sequences of characters enclosed by quote marks are called string constants, and are constants of the pre-declared type string.
> ⟨string constant⟩ ::= '{⟨character⟩}'
> Note 1: If the string is to contain a quote mark, then this quote mark is to be written twice.
> Note 2: The empty string constant consists of two successive quote marks.
> In appropriate contexts, string constants may be coerced to be constants of the standard type char (see 6.1.2) if they consist of a single character, or constants of the types
> **packed array** [1..*n*] **of** char
> if they consist of n ($n \geq 1$) enclosed characters (see 6.2.1). Note that this coercion occurs only for string constants, *not* for variables or values of type string.'

This definition will handle practically all programmer intentions in a simple manner, even anonymous comparisons like

if *'CAT'* > *'MOUSE'* **then** ...

A few expressions would be handled differently (with the same result); for example, the following statement

if '*A*' > '9' **then** ...

would involve the comparison of two one-character strings, whereas according to the existing Report it involves the comparison of two characters. A good optimizing compiler, however, may make the transformation, or indeed may evaluate the expression at compile-time. Appendix B discusses some of the reasons for rejecting alternative formulations of string constants and their handling.

5. PRE-DECLARED PROCEDURES

The foregoing rules for a pre-declared type based on the underlying sequence abstraction suffice to give a type of considerable utility, which yet conforms to rigorous axioms.[4] However, although these changes are sufficient, it is desirable (for efficiency reasons) to have a few pre-declared procedures which carry out commonly needed operations on strings. These procedures may be described in standard Pascal (thereby defining their semantics), but an implementation may well implement them in a different manner from the description.

A first useful function is one which returns the length of a string

```
type
  natural = 0 .. maxint;  {frequently used types}
  cardinal = 1 .. maxint;
function length(s : string) : natural;
  var i : natural;
  begin
    reset(s); i := 0;
    while not eof(s) do begin
      get(s); i := i+1;
    end;
    length := i;
  end;
```

Since users may construct their own procedures having string parameters, all that is needed is to provide a small basic set of pre-defined procedures which can be regarded as primitives. The following brief descriptions show a set which could be so regarded; definitions in terms of standard Pascal can be found in Appendix A. The character positions in a non-empty string are defined by putting the characters into one-to-one correspondence with the cardinal numbers (1, 2, 3, ...).

procedure *append*(**var** *s1* : *string*; *s2* : *string*);
{*the string s1 is extended by copying s2 onto its tail*}

procedure *extract*(**var** *dst* : *string*; *src* : *string*;
from : *cardinal*; *lngth* : *natural*);
{*the substring consisting of the characters in src starting at the position 'from' and continuing for 'lngth' characters are transferred to dst and replace its previous contents*}

procedure *insert*(**var** *dst* : *string*; *src* : *string*; *after* : *natural*);
{*the dst string has the characters of the src string inserted between the characters previously having ordinal positions* (*after*) *and* (*after* + 1). *If* (*after* = 0) *or* (*after* = *length*(*dst*)) *the insertion is equivalent to appending*}

procedure *delete*(**var** *dst* : *string*; *from* : *cardinal*; *lngth* : *natural*);
{*the dst string is modified by removing the characters whose ordinal positions are* (*from*) *to* (*from* + *lngth* − 1) *inclusive*}

function *find*(*s1*, *s2* : *string*) : *natural*;
{*the string s1 is searched for occurrences of the string s2 as a substring, and the function value returned is the ordinal number of the first character of such an occurrence. If no occurrence is found, or if s2 is empty, zero is returned*}

6. READING AND WRITING

Since strings are to be allowed as parameters in user-defined procedures and functions, and the type is a pre-declared one, it is necessary to examine the role played by strings in the read and write procedures which are defined on text variables. Firstly, examine the operation

write(*f*, *s*)

where *f* denotes a text variable, and *s* denotes a string variable. The obvious interpretation of this is that the characters in *s* are copied across to *f*. This neatly dovetails with

write(*f*, '*THIS IS A MESSAGE*')

and it can be seen that the parametric typing previously necessary no longer is: this operation is a write of a constant of type string.

The operation

read(*f*, *s*)

is more of a problem. However, recall that the line-marker in a file of type text is *not* in the type char, and therefore is unrepresentable in a string. The most appropriate interpretation is thus to transfer any characters on *f* into *s* up to, but not including, the next line-marker.

Formatted writes

write(*f*, *s* : *w*)

should be consistent with the existing rules of Pascal. If (*length*(*s*) < *w*) then the string *s* is preceded by (*w* − *length*(*s*)) spaces; if the string has length equal to *w* it is simply transferred; and if (*length*(*s*) > *w*) then the space allocated is expanded to *length*(*s*).

Interestingly, the same interpretation of string-constants as of type string given in the write example above legitimizes the CDC-specific pre-defined procedure *message*. This procedure, as defined in the CDC-specific part of the User Manual,[1] takes a parameter which is any-sized **packed array of** *char*. Now it can be seen as a respectable Pascal procedure having the heading

procedure *message*(*s* : *string*)

and thus conforming to all the strong typing rules.

7. FUNCTIONS

Since I have relaxed a number of rules which refer to the file structure, consideration should also be given to relaxing the prohibition on having functions of structured type. In the general case, this is a difficult step to take because it adds to the existing insecurities of Pascal without adding expressive power. The strongest argument against functions of any structured type is that it may be possible to write functions which return partially undefined values: results which have some components defined and others not. To add such an insecurity to Pascal would be indefensible; compilers could not easily detect situations in which this might happen. Therefore, since allowing functions of any structured type would only save the programmer from inventing a result variable to use with a corresponding procedure, they have not been allowed.

Would, however, functions of the specific type string make any sense? An obvious candidate to use the facility is the extract procedure shown earlier

procedure *extract*(**var** *dst* : *string*; *src* : *string*;
from : *cardinal*; *lngth* : *natural*);

which could be recast into function form:

function *extract*(*src* : *string*; *from* : *cardinal*;
lngth : *natural*) : *string*;

Given the likely frequency of use of strings, this would indeed be useful; and, as it turns out, the structuring properties of the sequence imply that it is *impossible* to have a partially defined result: files are always totally defined or totally undefined. Note also that an empty file is quite different from an unopened file (one on which no reset or rewrite has yet been issued). The only way of creating a file with undefined-value holes in it would be by *writing* an undefined-value to it, as in the following program fragment

$f\uparrow := 'E'$; *put*(f);
put(f); {$f\uparrow$ *is undefined, see Report* 10.1.1.}
$f\uparrow := 'D'$; *put*(f);

Usually the term 'undefined value' is taken to mean that use of this value is an error, and the above program fragment would be regarded as illegal. Some implementations check this interpretation at run-time.

Consequently, there is a quite defensible position from which one can argue that Pascal functions should be permitted to take on string type. Whether this is restricted to a few pre-defined functions, or extended to user-written functions, is a matter of judgement, not axioms.

There is also a counter-argument, which is based on arranging the types and structuring methods of Pascal into a sort of hierarchy, as shown below, with all types below a given level having some common properties

	file of ...
assignment	
	array of ...; **record** ... **end**
comparison	
	packed array[1..*n*] **of** *char*
expressions	
	set of ...
function values	
	real; *pointer types*; *integer*
index uses	
	Boolean; *char*; *enumerated types*; *subrange types*

If the ordering is accepted, allowing *string* (= **file of** *char*) to be a function value cuts right across the hierarchy: there are no expressions of string type. This argument is not conclusive and there are a number of holes in it. For example, there are not really any expressions of pointer type, nor is it obvious that the structuring methods of Pascal should be arrangeable into a linear hierarchy.

Therefore, without prejudice to the general issue of allowing structured functions, I conclude that functions of string type do not harm the general principles of Pascal, and their utility necessitates their inclusion. The extract procedure mentioned earlier has therefore been cast into function form in the Appendix.

8. ENCODING AND DECODING

Many users of non-standard versions of FORTRAN are familiar with the extensions known as ENCODE and DECODE (or INREAD and INWRITE), which essentially allow the FORTRAN programmer to use the formatting facilities to construct and interpret internal arrays of characters. Without conceding to the pressure from ex-FORTRAN programmers who want everything they had in FORTRAN to be in Pascal too, it can easily be seen from our new perspective that Pascal already implicitly has such facilities!

There is no requirement for a variable of file type to be attached to a named external object unless named in the program parameters; there is no requirement for part of its representation to reside on a mass storage device. All that *is* required is that it obeys the rules laid down for the sequence abstraction, and the practical additions in Pascal which have such things in mind. Consequently, a compiler could deduce that a particular file variable was never large enough to require disk space, or it could never claim disk space until a suitably sized internal buffer was full, or a compiler option (or a *pragmat*) could inform it that a particular file variable was intended to be represented in main memory. In any of these cases, the file is a true Pascal file, obeying all the Pascal axioms, and consequently reads and writes on it may do all the formatting things one would expect. As I noted before: the ENCODE and DECODE facilities are implicit.

The introduction of type string has a very simple implication for this viewpoint: if one were to invent a file-type for the purpose, it would look very like the string type. Consequently, programmers may use the read and write interpreting and formatting to assist in manipulating strings, or for any of the other genuine uses of the facility such as reinterpreting input lines.

The one problem with this use is that readln, writeln and eoln are not applicable to a file of string type since linemarkers are not representable in them. The possible ambiguity as to the meaning of

write(s, x)

is easily resolved. It is a write of x onto s (s being a file), and not a write of s and x onto *output*. The only lesson to learn is that the optional elision of the file-names input and output in Pascal is a regrettable feature.

9. IMPLEMENTATION AND PRAGMATIC VARIATIONS

Full implementation of string types requires dynamic memory management to maximize use of main memory, because the ultimate length of a string variable cannot be known. The necessary details are well known, and can be handled by any competent implementor, either by using the heap or some other memory allocation scheme (for example, a new segment per string variable on segmented-memory machines).

However, by limiting the abstraction ever so slightly, another simple implementation becomes possible which does not require dynamic memory management. This implementation is outlined here because it may be of use in micro-processor systems, or where no dynamic memory management scheme is available. It is clearly inferior to complete implementation conforming to the abstraction. The variation is to allow the programmer to state at compile-time the maximum size he expects the string variable to reach. The compiler then allocates sufficient space for this size. The only problem arises with procedure and function parameters of string type which are passed by a value mechanism; however, this case is identical to that of adjustable-bound arrays and can easily be resolved.

Clearly, the user-declaration of a maximum size is no part of the abstraction of a sequence, and it does not affect the rules applicable to string type except for one thing. If the number of characters in a string variable would exceed the limit at any time, the program is in error because it cannot conform to the axiomatic requirements, and it should be suitably terminated. Notice that because I have a clear idea of the abstraction to be modelled, the action on overflow is also clearly indicated: it is not truncation of the result, nor yet assuming the unrepresentable tail to be spaces, but *termination.*

This is effectively the implementation technique used in the UCSD compiler. Each association of string type to a variable is followed by a special piece of non-standard syntax (the vogue word is *pragmat*) to indicate the maximum length

```
var
  s1, s2 : string[80];
  pattern : string[212];
```

The details of this form of implementation are relatively easy to fill out. The structure which represents a string type must have a representation for the file window (buffer) and for the file status (open for reading, writing or closed). It also needs to be able to hold the actual length of the file at any particular time, and a maximum length. A possible structure could be as follows, where *maxstr* is the maximum string length:

```
type
  stringstructure = {by no means standard Pascal}
    packed record
      slength : 0..maxstr;
      status : (closed, readable, writable);
      sindex : 1..(maxstr + 1);
      s : packed array[1..maxstr] of char;
    end;
```

Using an extended Pascal as a documentation aid, it is then possible to describe the action of the pre-defined procedures (append, extract, length, etc.) in implementation terms, and similarly for the standard procedures reset, rewrite, and eof. For example,

```
procedure reset(ss : string alias sss : stringstructure);
  begin
    sss.status := readable;
    sss.sindex := 1;
  end;
procedure rewrite(ss : string alias sss : stringstructure):
  begin
    sss.status := writable;
    sss.sindex := 1;
    sss.slength := 0;
  end;
function eof(ss : string alias sss : stringstructure) : Boolean;
  begin
    if (sss.status ≠ closed) then
      eof := (sss.sindex > sss.slength)
    else
      terminatetheprogram;
  end;
```

10. CONCLUSIONS

This paper is easily organized into three parts. The prelude, in Sections 1–3, examines the properties of file structuring in Pascal, and lays the foundations for the central introduction of a pre-defined string type in Section 4. In the remaining sections (5–9) it is only necessary to trace out the logical consequences of the decisions made in Section 4.

The fact that the key decisions necessary to integrate strings of characters into Pascal can be kept in one small section is a convincing argument that the approach is a good one. The number of new concepts is thereby minimized, and an extended Pascal with these features is arguably no less well structured than a standard Pascal. Indeed, it can be argued that it is better structured, for several pre-defined procedures can be better described in terms of the extensions, and some restrictions on file types (necessary for pragmatic reasons) can be lifted for string variables.

I conclude that if any extensions to Pascal are going to address the string processing area, then the scheme outlined in the paper offers the best fit between problem and abstraction, and is founded on firm axiomatic principles. The alternatives, which include generalizing the array to have run-time variable bounds, lead to much greater damage to the strong typing principles of Pascal.

ACKNOWLEDGEMENTS

The ideas incorporated here owe much to many people. They were firstly provoked by the participants at the University of California Pascal Workshop in July 1978, and by contact with the UCSD implementation of Pascal. However, the stimulus for the effort arose directly from the forceful advocacy by Professor K. Bowles of UCSD for the absolute *necessity* of having some sort of facility for handling strings of characters in a Pascal framework. My thanks must also go to Bill Price of Tektronix for provocative comments which stimulated thought, to Jim Miner of the University of Minnesota for honest scepticism and to Judy Bishop for support.

APPENDIX A: DEFINITIONS OF PRE-DEFINED PROCEDURES

This appendix serves to define the semantics of the set of procedures which are suggested as a minimal set for handling string variables. The procedure *copy* is introduced to simplify the descriptions, but should be considered as hidden from the user. Implementations of the procedures should be semantically equivalent to these definitions

```
procedure copy(var s1 : string; s2 : string);
  {expects s1 to be open for writing, and copies s2 onto s1, leaving s1 in a state for continued
  writing}
    begin
      reset(s2);
      while not eof(s2) do begin
          s1↑ := s2↑; put(s1); get(s2);
      end;
    end;
procedure append(var s1 : string; s2 : string);
    var st : string;
```

```
    begin
      rewrite(st); copy(st, s1);
      copy(st, s2);
      rewrite(s1); copy(s1, st);
        {the file status and file window of s1 are left undefined}
    end;

function extract(src : string; from : cardinal; lngth : natural) : string;
    var i : cardinal;
      dst : string;
    begin
      if (length(src) < (from + lngth - 1)) then terminatetheprogram;
      reset(src); rewrite(dst);
      for i := 1 to (from - 1) do get(src);
      for i := 1 to lngth do begin
          dst↑ := src↑; put(dst); get(src);
      end;
      extract := dst;   {non-standard, but unavoidable}
        {the file status and file window of the result are undefined}
    end;

procedure insert(var dst : string; src : string; after : natural);
    var i : cardinal;
      st : string;
    begin
      if (length(dst) < after) then terminatetheprogram;
      reset(dst); rewrite(st);
      for i := 1 to after do begin
          st↑ := dst↑; put(st); get(dst);
      end;
      copy(st, src);
      while not eof(dst) do begin
          st↑ := dst↑; put(st); get(dst);
      end;
      rewrite(dst); copy(dst, st);
        {the file status and file window of dst are left undefined}
    end;

procedure delete(var dst : string; from : cardinal; lngth : natural);
    var i : cardinal;
      st : string;
    begin
      if (length(dst) < (from + lngth - 1)) then terminatetheprogram;
      reset(dst); rewrite(st);
      for i := 1 to (from - 1) do begin
          st↑ := dst↑; put(st); get(dst);
      end;
```

```
        for i := 1 to lngth do get(dst);
        while not eof(dst) do begin
            st↑ := dst↑; put(st); get(dst);
        end;
        rewrite(dst); copy(dst, st);
          {the file status and file window of dst are left undefined}
      end;

  function find(s1, s2 : string) : natural;
      var M, N : natural;
        i : cardinal;
        state : (scanning, found, notfound);
      begin
        M := length(s1);
        N := length(s2);
        if (N = 0) or (M < N) then begin
          find := 0;
        end else begin
          i := 1;
          state := scanning;
          while (state = scanning) do begin
            {The following if uses a non-standard string comparison and a string function
            defined earlier}
            if (extract(s1, i, N) = s2) then begin
              state := found; find := i;
            end else begin
              i := i+1;
              if ((M-i+1) < N) then begin
                state := notfound; find := 0;
              end;
            end;
          end; {of while loop}
        end; {of if testing lengths}
      end; {of find}
```

APPENDIX B: JUSTIFICATION OF SOME DECISIONS

Undefinition of the file state

In this paper, it is stated that the file state information associated with a string variable was left undefined after every string operation: assignment, value parameter passing, comparison, etc. Why ? Why should $S2$ have its state undefined in

$S1 := S2$?

The conflict arises basically from the difference between the string operations defined in the paper which treat the string as an *entire object*, and the get and put operations which operate on the detail structure of the file. Abstractly, the entire operations might define a state for the detail, or they might do as I suggest: leave the state undefined.

The problems with defining the state after all string operations are not trivial. Are some operations not to disturb a state ? The right-hand side of an assignment, the actual argument

to a value parameter, or a string involved in a comparison, for example ? In what state are assigned strings to be left ? Writeable and at end-of-file, or exact copies of the source ? The choice between these possibilities is difficult; there are good arguments for each particular course of action.

However, it seems to me that this is over-specification: entire-variable operations should not be involved or intermixed with substructure fiddling. Making the state information (read/write/open status and the file window positioning) undefined for all entire-variable uses of files neatly makes the problem disappear. A programmer may not then insert entire-variable operations into the middle of some structure fiddling without resetting or rewriting subsequently.

String constant coercion

At an early stage of these investigations, it was suggested that there should be no constants of the string type, but that there be an automatic coercion from char type, and from packed array of char generic types into string type. This suggestion was rejected, together with some other variations, and it is important to see why.

Probably the suggestion arose from a desire to leave the Revised Report in its original form as far as possible, and this it does. The coercion is simply an extension. However, a general principle of automatic coercions that is now widely accepted, and is incorporated in Pascal's design, is that they only be permitted when the coercion implies a widening of the abstraction. Thus coercing integers into real numbers, or real numbers into a complex number domain, are reasonable. The suggestion does not meet this criterion. A string can be seen as an extension of the char type without too much difficulty, but it cannot easily be seen as an extension of an array type. If anything the array type is more highly structured than the sequence type, in that its sequencing is implicitly defined by its indextype, as are its bounds and accessing methods.

Structure is therefore lost when coercing an array to a sequence. In the general case, of course, an array may not be sequenced at all if its indextype was an unordered scalar type; this shows the basic abstraction of an array as a mapping from one domain to another very clearly.

Objectively, the lexical token in Pascal which expresses a series of characters is a sequence: it has no way of specifying any array properties except implicitly. Consequently, the initial decision taken in Pascal to define this token as a constant of char and the packed array of char types can be seen to simply be a consequence of the lack of a string type, and less natural.

In defining string constants in Section 4, several different formulations were tried, with the intention of preserving the existing uses of the token as well as the new one. These included context-dependent types, as well as the result actually in that section. The form finally adopted has the advantage that it limits the decisions to ones which can all be made at compile-time, and a by-product of clarifying uses of string-constants of one character. It also clarifies the position of string constants used as actual parameters to pre-defined procedures.

REFERENCES

1. K. Jensen and N. Wirth, *Pascal—User Manual and Report*, Springer-Verlag, Berlin, 1975.
2. O. J. Dahl, E. W. Dijkstra and C. A. R. Hoare, *Structured Programming*, Academic Press, New York, 1972.
3. N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, New Jersey, 1975.
4. C. A. R. Hoare and N. Wirth, 'An axiomatic definition of the programming language Pascal', *Acta Informatica*, **2**, 335–355 (1973).