University of Glasgow
Department of Computing Science

Lilybank Gardens
Glasgow G12 8QQ

University of St Andrews
Department of Computational Science

North Haugh
St. Andrews KY16 8SX

Applications Programming in PS-algol.

Richard Cooper

PPRR 25.

Alan Dearle.

Applications Programming In PS-algol.

Richard Cooper

Table Of Contents.

7/ Interactive Compilation.

0/ Introduction.

This document is intended to introduce the reader to some of the techniques found useful for writing programs in PS-algol, with particular emphasis on the following features of the language:

- the provision of a persistent store;
- the graphics facilities;
- the availability of first-class procedures;
- the existence of a universal pointer type;

and • run-time compilation

The persistent store simplifies data storage. The method of providing persistence will be described in brief and a mechanism for structuring updates as atomic transactions will be shown. As an aside, the method of access to the underlying file store will also be discussed.

The graphics facilities of PS-algol not only allow graphical data to be stored in a database with the same ease as text and numbers but also provide a set of operations with which user interfaces can be constructed. The construction of an error message facility, a forms interface and a simple string editor will be used to illustrate the way in which these facilities can be used.

First-class procedures provide a method for making functional abstractions. Procedures may be manipulated in the same way as other objects within the language and so may be used to:

- implement modules of the program which can be compiled separately and linked together via the persistent store;
- model actions, which can then be stored in the same way as simple data types, such as numbers and strings;
- hide the structural and representational details of an object model by making creation procedures return data structures containing only procedures. These will provide the operations available on the object, thus giving the facilities of Abstract Data Types.

The use of the persistent store to maintain a procedures library systematically and the method of developing a large program in a modular fashion will be described. Some examples of using procedures to model actions will be given and the method of creating objects as Abstract Data Types will be shown.

The pointer type of PS-algol permits the construction of object structures of arbitrary complexity, which makes PS-algol a flexible tool for modelling objects. We will show how, for instance, an Entity-Relationship data model can be modelled. We will also show how the type **pntr** allows the type checking of the binding of the program to its data to be deferred until the last possible minute.

The availability of a version of the PS-algol compiler as a standard function of the language allows a program to construct, compile and run procedures during its run. We will show how this allows us to write truly polymorphic procedures within a strictly type-checked environment. It also allows us to write programs which permit the binding of an already written program to data of a newly introduced type. This mechanism is more flexible than using the pointer type, since the programmer needs to know nothing about the newly introduced type. It is also more efficient, since the resulting procedures will access the data directly, instead of via a series of indirections.

The use of these facilities, and their interaction, is illustrated in the remainder of this report by means of a series of examples which have been written in a programming style found to be effective in PS-algol. PS-algol is a relatively new language and probably the first which allows data structures to persist between program runs. Therefore the programmer is not required to organise transfers between database and active heap data, yet allows new programs to access pre-existing databases. The novelty of this orthogonal persistence leads to new styles of programming. The appropriate style for exploiting these opportunities is not yet known, that presented here being only a first approximation to the optimal style. The reader is encouraged to carry the development further.

The language is not defined in this document and the reader should refer to the PS-algol reference manual (Third Edition) [PPRR12]. For an introduction to the language, the reader is referred to [PPRR31]. What are given here are expansions on some of the more unusual features of the language, with new examples to illustrate some of the less obvious problems. The various uses of first-class procedures were described previously [ATKI85].

1/ Building Complex Data Structures.

a) Introducing Structures and Pointers.

An unusual and powerful feature of the PS-algol type-system is the **pntr** type. This was introduced as a method of modelling a structure of arbitrary complexity. A complex object class is introduced with a **structure** statement. For instance, we could define a structure to hold the elements of a list of strings by

```
structure StringListElement( string value; pntr next )
```

which means that any element of such a list has two fields, a **string** field containing the value of the element and a **pntr** field pointing to the next element in the list. Note that the structure name, *StringListElement*, and the field names, *value* and *next*, must be uniquely used identifiers within the block in which the structure is defined. Creating a complex object is achieved by using the structure name as a class constructor, for instance:

```
StringListElement( "A", AnotherElement )
```

creates an element containing the value "A" and a pointer to another object. This other object can, in fact, be an instance of any class introduced by the **structure** command, although in this case we would expect it to point to another *StringListElement* or to the pointer constant, **nil**, to indicate the end of the list.

As an example of how this is used, let us define a variable to hold our, initially empty, list by the command

```
let StringList := nil
```

and then define a procedure which adds a new value, *NewValue*, onto the front of the list, as follows:

```
let AddStringToList = proc( string NewValue )
        begin
                StringList := StringListElement( NewValue, StringList )
        end
```

The first application of the procedure, for instance by the call:

```
AddStringToList( "first" )
```

will construct an element with field values "first" and **nil** and put it into the list by making *StringList* point to it. A second call will create a new element, pointing to the first, and set *StringList* to point to it, thus putting the new element onto the front of the list. (Note that PS-algol does not really require the **begin...end** of the procedure *AddStringToList* to be put into the program in the case where the procedure body consists of a single clause. They have been included here only for clarity).

The fields of a structure are dereferenced by giving the object name followed by the field name in parentheses. Thus the first value on our string list is accessed by

```
StringList( value )
```

To illustrate, here is a procedure which scans and prints the values of the list:

```
let PrintStringList = proc()
        begin
                let P := StringList
                while P ~= nil do { print P(value); P := P(next) }
        end
```

The pointer variable, *P*, traverses the list, successively accessing the *value* field to print it and the *next* field to move on to the next item of the list. Note that the curly brackets have been used as synonyms for **begin** and **end** for brevity.

b) Modelling Complex Structures.

Since PS-algol is data type complete, the fields of the structures can be of any type and so give us the ability to model complex data structures. For instance, we can model:

- numeric and textual data in fields of types **int**, **real** and **string**;

- graphical data with the types **pic** and **#pixel**;

- actions using procedures;

and • more complex objects, such as the list we have just seen using pointers to other objects.

We will later (sections 3c, 6c and 6d) show how to construct a forms interface. The form has been modelled as a list, the basic element of which is the light button. This is a rectangular area of the screen on which is displayed a message in a box . When the mouse is clicked over that area some associated action is performed. This is modelled by the following structure:

```
structure button(    string message;                ! the displayed message
                     int xo, yo, xh, yh;            ! the position and dimensions of
                                                    !   the screen area
                     #pixel save;                   ! the contents of the area, before
                                                    !   button was displayed
                     proc() action;                 ! the associated action
                     pntr next      )               ! the next button in the list
```

For a further example, we take the Entity-Relationship-Attribute model similar to the one proposed by Chen [CHEN75]. In this model, objects are modelled as "entities", which have properties called "attributes" and are linked to other objects via "relations". Figure 1 shows a general view of what such a model might look like. The circles represent three entity types, "A", "B" and "C". Associated with each is a rectangle showing the attributes defined on it, their names in italics and type in bold. The relations are represented by lines connecting the circles.

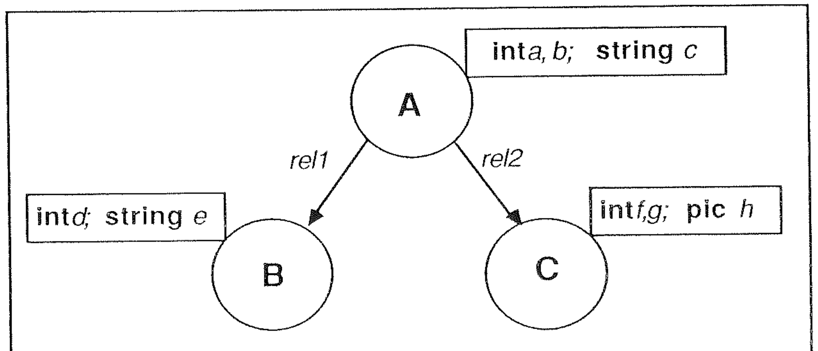


Figure 1. An Entity-Relationship Data Model

This may be modelled in PS-algol as follows:

```
structure A( int a,b; string c; pntr rel1, rel2 )
structure B( int d; string e )
structure C( int f,g; pic h )
```

with the entities being modelled by structures, the attributes being modelled by scalar fields of the structures and the relationships being modelled by pointer fields linking the structures.

A concrete example of this model in practice is the schema for a simple students and courses database, shown in Figure 2.

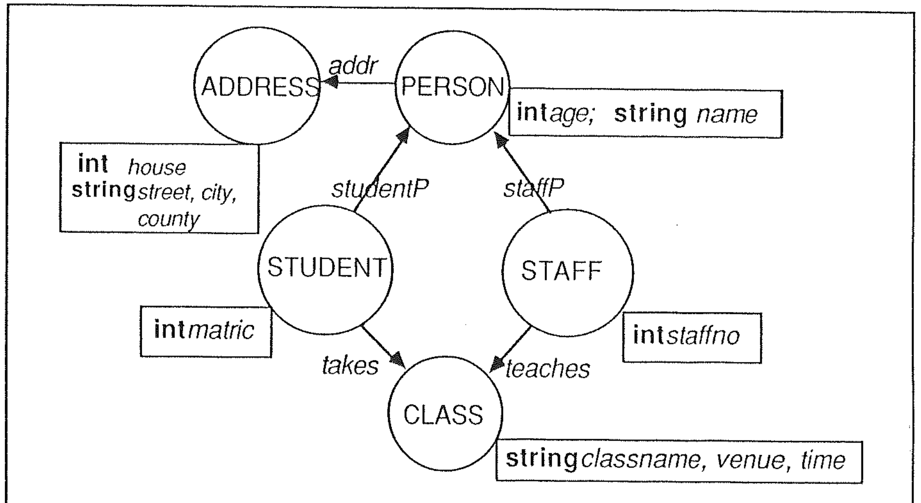


Figure 2. An Entity-Relationship Model of a Student Database

This might be modelled by:

```
structure PERSON( string name; int age ; pntr addr )
structure ADDRESS( int house; string street, city, region )
structure STUDENT( int matric; pntr studP; *pntr takes )
structure STAFF( int staffno; pntr staffP; *pntr teaches  )
structure CLASS( string className, venue, time )
```

Notice here that one-to-one relationships are modelled by fields of type **pntr**, while one-to-many relationships are modelled by fields of type ***pntr**. Actual objects in the system will be modelled by instances of the structure classes. For example, a student will be modelled by a *STUDENT* structure, together with an associated *PERSON* structure. The model therefore incorporates a notion of inheritance as all of the information known about the student can derived either from the *STUDENT* structure or, by following the pointer field *studP*, from the *PERSON* structure.

c) Partial Polymorphism via Pointers.

PS-algol is a strictly typed language. Any attempt to supply a value of one type where another is expected will result in a compile-time error. In particular, it is necessary to specify the types of the parameters of a procedure. It is not then allowed to supply parameter values of another type, even when it is possible to specify a meaning for the procedure with the second type. It is not permitted, for instance, to supply two reals to a procedure which adds together two integers. This restriction results, for instance, in the unsatisfactory state of having two sets of table manipulation procedures, one for string keys (**s.enter**, etc) and one for integer keys (**i.enter**, etc).

However, one can achieve a restricted form of polymorphism through which procedures receive their input values via pointers to packages containing them. To illustrate this technique, let us return to our list example and extend it to include objects of a number of types. We now define our elements to be of the type:

```
structure AnyListElement( pntr value, next )
```

that is the value field is also a pointer field and also define some packaging structures:

```
structure StringPack( string Svalue )
structure IntPack( int Ivalue )
```

etc.

We will initialise the list with

```
AnyList := nil
```

and the insertion procedure becomes:

```
let AddAnyToList = proc( pntr NewValue ); AnyList := AnyListElement( NewValue, AnyList )
```

(Note that the **begin....end** bracketing has been dropped as the procedure body consists of a single clause). Our initial insert is now slightly more complex, as follows:

```
AddAnyToList(StringPack( "first" ) )
```

where an object of class, *StringPack*, containing "first", is created and then inserted into the list. We could then follow this, by inserting the integer, 2, as the second element by:

```
AddAnyToList( IntPack( 2 ) )
```

Thus *AddAnyToList* is a procedure which takes no account of the type of data which it is handling. That information only becomes important when writing procedures which scan the list and use the values. In writing such a procedure, it is necessary to know what classes of objects are in the list. Thus we can re-write the printing procedure to handle the case where only integers and strings have been stored as follows:

```
let PrintAnyList = proc()
        begin
                let P := AnyList
                while P ~= nil do
                      begin
                            let V = P( value )
                            if V is StringPack do print V( Svalue )
                            if V is IntPack do print V( Ivalue )
                            P := P(next)
                      end
        end
```

Note that objects of any other classes will just be ignored and if other classes are added to the system, more lines of the form "if V is ..." will have to be put into the procedure and it will have to be recompiled. In order to write a truly polymorphic procedure, the program must be organised to create print procedures for structure classes as it encounters them and to compile them as the program runs. Section 7 goes into more detail on how to go about this.

To sum up, the **pntr** type of PS-algol allows objects of different types to be stored in a common way. The binding of the program to the values of the objects is deferred until the values are dereferenced. However, the method we have been using forces the programmer to determine which classes of object the program expects to encounter in those modules which use the data.

d) Difficulties With Pointers.

Two main difficulties have to be overcome when handling the pointers and structures of PS-algol. Sometimes the values of objects change when no change is intended and sometimes an intended change does not occur.

The former case often occurs when making a copy of an object. Let us say an address has been constructed as follows:

```
structure ADDRESS( int house; string street, city, region )
let CSglasgow:= ADDRESS( 17, "Lilybank Gdns", "Glasgow", "Strathclyde" )
```

and we now do

```
let ITglasgow := CSglasgow
```

This makes *ITglasgow* point to the same object as *CSglasgow*. It **does not** make a new object with the same values as *CSglasgow*. Therefore, if the values of the fields of *ITglasgow* are now changed, then as *CSglasgow* points to the same object, the fields of *CSglasgow* are changed as well. Thus

```
ITglasgow(city) := "Pisa"
```

makes *CSglasgow* be *ADDRESS*(17, "Lilybank Gdns", "Pisa", "Strathclyde"), just the same as *ITglasgow*. Note that the command:

```
ITglasgow := ADDRESS( 17, "Lilybank Gdns", "Pisa", "Strathclyde" )
```

does not have the same effect. This introduces a completely new referend for *ITglasgow* and leaves *CSglasgow* pointing to its original value.

In general then, if you require *CSglasgow* and *ITglasgow* to refer to completely different objects, you must construct a new object for *ITglasgow* to point to. The fields of this new object will be copies of the fields of *CSglasgow*. Furthermore, it is not enough to take copies only at the topmost level of the object. New objects must be introduced at every level of the object hierarchy. Thus if we have

```
structure PERSON( string name; pntr addr )
structure ADDRESS( int house; string street, city, region )
let RC:= PERSON( "Richard Cooper",
              ADDRESS( 17, "Lilybank Gdns", "Glasgow", "Strathclyde" )   )
let XX := PERSON( RC( name ), RC( addr ) )
```

The first fields of *RC* and *XX* have now been completely separated, but the second fields both point to the same object and any changes on the sub-object of that field will affect both, i.e.

```
XX( addr )( city ) := "Pisa"
```

affects *RC* as well as *XX*. You must do, in this case,

```
let XX := PERSON( RC(name ),
              ADDRESS( RC(addr)(house), RC( addr )( street ) ....... )    )
```

to be sure that all sub-objects are distinct.

Problems of the second kind, not getting changes when they might be expected, are encountered in the following type of situation. Let us assume that you have a global set of variables that you wish to share between program modules - for instance:

```
structure Globals( int sum, product )
```

which is defined in two different modules, manipulated by them and stored in the database. Thus

```
G := s.lookup( "theGlobals", db )
```

is used to get them out. Then you do

```
theSum := G( sum )
```

and use *theSum* throughout to make the program more readable and efficient. This makes an entirely new object and so if you don't rememember to do

```
G(sum) := theSum
```

before leaving the procedure, then all changes to the value will be lost. If in fact you make only a little use of *theSum*, then it may be wiser to use *G(sum)* throughout anyway.

e) Difficulties with Vectors.

PS-algol vectors give rise to two sources of difficulty: the fact that their size is not dynamically variable; and the fact that the **vector** notation for introducing them causes all elements of a vector to point to the same object.

Let us say that we have a vector containing a set of addresses. The most obvious way of writing a procedure to insert an address into such a vector might be as follows:

```
let addressVector = vector 0 :: 0 of nil            ! standard method for overcoming
```

```
let insertAddress := proc ( pntr newAddress )          ! the lack of zero length vectors
    begin
        let NumAddresses = upb(addressVector )
        let tempVector= vector 1:: NumAddresses  + 1 of nil
        for i = 1 to NumAddresses do tempVector (i) := addressVector (i)
        tempVector ( NumAddresses  + 1 ) := newAddress
        addressVector := tempVector
    end
```

This has little to recommend it as it is creating vectors of each size 1, 2, etc and so making a lot of large objects. This is slow in itself, and will also cause the garbage collector to be activated more often than necessary - a further slowing. One way round the problem is to create a temporary vector large enough to hold the most entries that can be conceived of is used and then strip this down to the right size at the end. If a largest size cannot be guessed at, the size of the temporary vector could be increased by a 100, say, when it becomes filled. An alternative method is to create the set of objects in a more flexible structure, such as a list, and then copy the set into a vector at the end. In any case, vectors to not lend themselves to data structures where element insertion and deletion is frequent.

To introduce the second problem consider the following declaration:

```
let intVector = vector 1::10 of 0
```

All ten elements of the vector contain the value of the same expression, 0. This does not cause any problem for one dimensional vectors, as each element may then be re-assigned, without affecting the others. For instance:

```
intVector( 3 ) := 7
```

only changes the third element of the vector as expected. However, in the case of vectors of higher dimensions - for instance

```
let doubleVector = vector 1 :: 10 of vector 1 :: 5 of 0
```

we have here set up a vector whose 10 elements are all the value of the same expression which is a vector of 5 zeroes. If we now re-assign the value of the element with the largest indexes, for instance by:

```
let doubleVector( 10, 5 ) := 7
```

then this changes the fifth element of the vector all ten elements of the top level vector are pointing at. Therefore, the above command also changes to 7 the values of *doubleVector*(1, 5), *doubleVector*(2, 5), etc. This thoroughly undesirable event can only be circumvented by following the declaration line by the command:

```
for i = 1 to 10 do doubleVector( i ) := vector 1::5 of 0
```

which has the effect of creating a different sub-vector for each element of the top-level vector. Note that both the declaration and the assignment lines are necessary.

2/ Problems With I/O.

a) Keyboard Input.

As an aside at this point, it is useful to discuss the use of the keyboard input functions in the PERQ implementation as these give rise to some complications. If you just plunge in and use all the commands given in section 11.1 of the reference manual, the input characters are echoed on the screen via standard output, in the same way that **write** displays its output. In common with **write**, this will probably have the effect of scrolling the screen and making a mess of your display. Another problem is that no input characters will be passed to the program until the return key is pressed. Any control characters will by then have been stripped off and perhaps acted upon by the system. Essentially only **read.a.line** functions effectively. What is required is a method which immediately passes the characters to the program, without waiting for "return" or echoing them. This is achieved by placing the following command before using one of these commands:

```
let set = system( "stty -echo cbreak raw" )
```

This makes a call to PNX, which stops echoing and causes characters to be immediately passed back to PS-algol from the keyboard, without waiting for the return character and without filtering out control characters.

This call changes the screen handler for this window, which is part of the underlying system and is not part of PS-algol. Therefore, when the program is quit the effect of the line above will persist so that further commands to PNX are not echoed - you can't delete characters before pressing return, etc. Therefore, before the program quits, you must do:

```
let reset = system( "stty echo ~cbreak cooked" )
```

to undo what you've done. It is a matter of style whether you just put one *set* at the top of your program and a *reset* at the bottom or bracket each **read** with a pair of *set* and *reset*. In any case, it is a good idea to have a *reset* file in your bin directory ready to reset a window when the program crashes.

There are also some problems with the implementation of the various **read** commands. It seems that the sequence:

```
x := read.int()
y := read.a.line()
```

will always set *y* to the empty string, as the return character which terminates the **read.int** is not cleared and so it also terminates the **read.a.line**.

b) Combining Keyboard and Mouse Input.

The system function, **input.pending**, returns a boolean whose value is **true** if a keyboard key has been pressed. This can be used to mix the input of responses from the mouse and the keyboard as follows:

```
let mixedInput = proc( -> string )
  begin
     let waiting := true
     let c := 0
     let maxwell := nil
     while waiting do
          begin
               if input.pending() do { c := read.byte(); waiting := false }
               if waiting do
                    begin
                         maxwell := locator( )
                         for i = 1 to 4 do
                              if maxwell( the.buttons ) ( i ) do { c:= -i; waiting := false }
                    end
          end
     c
  end
```

This procedure stays in its inner loop until the *waiting* flag is set and then returns a result via the variable, *c*. The flag is set either if a key on the keyboard is struck (in which case *c* is set to the ASCII value of the appropriate character) or if a mouse button is pressed (in which case *c* is set to -1, -2, -3 or -4 depending upon which button was pressed).

c) File i/o.

Files have been a closely guarded secret within the PS-algol community. One of the main goals of the Persistent Programming Research Group is to eliminate the need for files and file managers altogether. However, it is recognised that present implementations sit within a file system and it is essential to have the ability to pass information to and from that file system.

Outputing to a file requires the commands **create** and **output** as follows:

```
let f = create( "Afile", 511 )
```

creates a file in the current directory called *Afile* with UNIX file permissions 777 (octal) = 511 (decimal). The variable *f* is of type **file** and is then used, for instance, in:

```
output f,"a line of text'n"
```

where **output** behaves exactly like **write** except that it also has an extra parameter - the file to be written to. Files are automatically closed at the end of a program, but may also be closed during the run by

```
close( f )
```

A file is opened for input, by the command

```
let f = open( "Afile", 0 )
```

where the second parameter specifies a mode, which is 0 for read access, 1 for write access and 2 for both. Again *f* is of type file. All the usual input commands can then be switched to a file, by putting the file descriptor as a parameter. Thus

```
let x = read.byte()
```

reads a byte from the keyboard, whereas

```
let x = read.byte( f )
```

reads it from the file.

We present a set of procedures which have been found useful for interfacing to the file system as Appendix A. The output is spooled in a string vector by procedure *putLine* and sent to the file at the end of the program by a call to *putFile*. Conversely, the input is read into the input spool, *thelfile*, by the procedure *getFile*, at the start of the program and then individual lines of text are read into the program by the procedure, *getLine*. By organising the i/o in this way, the program does not have to continually access the file system. Clearly, if a large file is being output this may not be the best method as having a large string vector may be too high a price to pay.

3/ Creating User Interfaces Using The Graphics Facilities.

a) Images and Screen Manipulation.

Objects of type **#pixel** are rectangular bitmaps and fall into two categories. Base bitmaps are introduced by the **image** constructor, for instance:

```
let baselm = image 12 by 12 of off
```

while images introduced with the **limit** constructor are aliases of part of another image. For instance, the command:

```
let quad = limit baselm to 6 by 6 at 0, 6
```

creates an object which is aliased to the top left-hand quadrant of *baselm*. Any changes to the top left quadrant of *anImage* will simultaneously be made in *quad* and vice-versa. Figure 3 shows this diagrammatically.

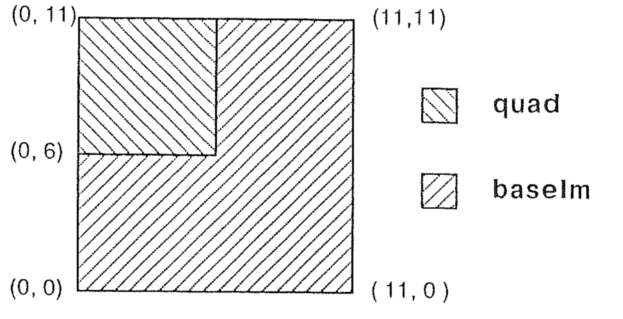


Figure 3. Base and Aliased Images

screen is the name of a base bitmap which is permanently mapped onto the currently open screen window. Its dimensions are fixed at the time the program are started and if a window manager function is used to change the size of the screen during a program run, the program will abort with a run-time error. The screen is cleared with the command

```
xor screen onto screen
```

This is a standard technique for setting all of the pixels of an image to **off**. They may equally be set to **on** by the command:

```
xnor screen onto screen
```

which will make the screen completely black.

A rectangular area of the screen may be referenced using the **limit** function to create an alias bitmap, e.g.

```
let area = limit screen to xsize by ysize at xorg, yorg
```

Now any bitmap operations on *area* show through onto that area of the screen and so

```
xnor area onto area
```

blacks out that area of the screen. NB: the latter is always faster than

```
xnor limit screen to xsize by ysize at xorg, yorg
        onto limit screen to xsize by ysize at xorg, yorg
```

as the latter has to construct the *area* image twice and, as we have seen for vectors, this both takes more time to construct the image and causes the garbage collector to be invoked more often than necessary.

b) Constructing An Error Message Facility.

The following is a procedure which draws a box in a given position of width two pixels:

```
let box = proc( int xorg, yorg, xsize, ysize )
        begin
                let area = limit screen to xsize by ysize at xorg, yorg
                let inner = limit area to xsize-4, ysize-4 at 2, 2
                xnor area onto area                     ! make area black
                xor inner onto inner                    ! make inner white
        end
```

Two aliased images are defined which refer to the whole of the area of the box and the area within the border. The outer area, *area*, is blacked out and then the inner area, *inner*, is cleared to white. Note that *inner* is defined relative to *area* although it could have been directly defined as

```
limit screen to xsize-4 by ysize-4 at xorg+2, yorg+2
```

Another useful procedure is the following, which waits until the top button of the mouse is pressed and released, which events are recognised by calling **locator** and checking if the first element of the *the.buttons* field is set first to **true** or **false** :

```
let PressRel = proc( )
  begin
        let maxwell := locator()
        while ~maxwell( the.buttons )(1) do maxwell := locator()  ! wait on mouse button
        while maxwell( the.buttons )(1) do maxwell := locator()
  end
```

We can now build these into a procedure which shows an error message in a box at a given point of the screen and waits for the mouse to be pressed and released before restoring the original screen contents:

```
let ErrorMessage = proc( string message; int xorg, yorg )
  begin
        let MessageImage = string.to.tile( message, "fix13" )
        let xsize = X.dim( MessageImage ) + 10
        let ysize = Y.dim( MessageImage ) + 10
        let area = limit screen to xsize by ysize at xorg, yorg
        let save = image xsize by ysize of off
        copy area onto save
        let inner = limit area to xsize-4, ysize-4 at 2, 2
        xnor area onto area                             ! make area black
        xor inner onto inner                            ! make inner white
        copy MessageImage onto limit area at 5,5        ! put the mesage into the box
        PressRel( );    copy save onto image
  end
```

There are several points to note here. The area size is calculated from the size of the image to be displayed (+ a border of 5 pixels). The image object *save* is used to store the screen contents that will be overwritten when the message is displayed. Note that *save* is not a part of the screen, so any operations on it will not be visible. After the box has been constructed, the message is put into it. After *PressRel* finishes, the original screen area is restored.

c) A Simple Form System.

Let us now try to develop a set of procedures which put up a set of rectangles on the screen any of which may be pointed to by the mouse to initiate some operation - we will call these "light buttons". In this first attempt, we will just provide the ability to determine which message has been pointed to, by returning the message itself.

We need to have a list of the buttons and an initiating procedure:

```
        let ButtonList := nil
and     let ClearButtonList = proc(); ButtonList := nil
```

We then need a structure for our buttons and a facility for adding a new one:

```
        structure button( string StMess; int xo, yo, xs, ys; #pixel SavedArea; pntr NextButton )
and     let NewButton = proc( string message; int xorg, yorg )
            begin
                let MessageImage = string.to.tile( message, "fix13" )
                .............................................................................
                .... as for ErrorMessage procedure above ....
                .............................................................................
                copy MessageImage onto limit area at 5,5
                ButtonList := button( message, xorg, yorg, xsize, ysize, save, ButtonList )
            end
```

Finally, we need a monitoring procedure to return the button pointed at:

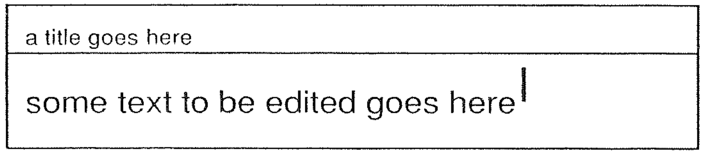
```
let MonitorButton = proc( -> string )
      begin
            let return := ""
            while return = "" do
               begin
                  let maxwell := locator()
                  while ~maxwell( the.buttons )(1) do maxwell := locator()
                  while maxwell( the.buttons )(1) do maxwell := locator()
                  let X = maxwell( X.pos )
                  let Y = maxwell( Y.pos )
                  let P := ButtonList
                  while P ~= nil do
                   {  if P(xo) <= X and X <= P(xo)+P(xs) and
                         P(yo) < Y and Y <= P(yo)+P(ys) do return := P( StMess )
                         P := P( NextButton )          }
               end
            return
      end
```

The procedure waits until the mouse button is pressed and then scans through the list of buttons, checking to see if the mouse is currently within the area of each. When it finds such a button, it stores the associated message in the variable, *return*, and the value of this is returned by the procedures.

We will return to this example in later sections to illustrate the way in which first-class procedures enable us to make the form more efficient and powerful.

d) A Simple String Editor.

Our final example shows the code for a simple string editor. The procedure takes in a title, an initial string value and the origin and dimensions of a box to do the editing in. The box is drawn with the title at the top in a small font and the initial value inside the box in a large font. Figure 4 shows the editor in action.


a title goes here

some text to be edited goes here

Figure 4. A Simple String Editor

The procedure responds to:

- the mouse to move the cursor about the string;
- the del key to delete the last character;
- the oops key to delete everything to the left of the cursor;
- return to quit the editor;
- all other control characters are ignored;

and • any printing character is added to the string at the current cursor position.

The procedure, which is given as Appendix B, starts by displaying two boxes and putting the title in the upper box. The following are then created:

- an image constant which contains the cursor as a short vertical line;
- two string variables, *leftText* and *rightText*, which will contain those parts of the string to the left and right of the cursor;

and • an integer, *cursorPosn*, to contain the current cursor position as the number of characters to the left of it.

Then follow two procedures:

- *CursorDisplay*, which returns the part of the screen where the cursor should currently be;

and • *showText* which will update *CursorPosn*, *leftText* and *rightText*, and then display the new configuration.

The main part of the procedure starts by calling *showText* to initiate the display with the cursor at the right hand end of the input text. The bulk of the procedure consists of a loop controlled by the boolean, *exit*, which is set when the return key is pressed. The loop uses a reduced version of the technique described in section 2b above to get input either from the first mouse button or from the keyboard, whichever is given first. The variable, *c*, is used to hold the resulting input. Its value is either -1 if the mouse button is pressed or the ASCII value of the character if a key on the keyboard is pressed. A case statement then controls the action as follows:

- if the mouse button is pressed within the box, then the cursor is re-positioned by a call to *showText* ;
- if "oops" is pressed, the text to the left of the cursor is erased from the screen and from *leftText*;
- if delete is pressed, the character to the left of the cursor is erased;
- if return is pressed, the exit condition is set;
- any other control character is ignored;

and • if a printing character is pressed, it is inserted into *leftText* and the whole string is re-displayed.

Finally the procedure restores the screen and returns the concatenation of *leftText* and *rightText* as its result.

4/ The Persistence Mechanism and Database Organisation.

a) Tables.

The main organising structure provided by PS-algol is the **table**. This is a collection of key/value pairs, in which the value is an object of type **pntr**, that is an object created by a class constructor. The key can be either a string or an integer, although we will restrict our discussion here to string keys. We create a new empty table with the **table** command, for instance:

let *addressTable* = **table**()

and insert new entries with the **s.enter** command:

s.enter("CS", *addressTable* , *ADDRESS*(17, "Lilybank Gdns",...))

A value is retrieved by the command:

```
let CS = s.lookup( "CS", addressTable )
```

which returns a pointer to the value associated with the key, "CS" - in this case, the *ADDRESS* structure above. If the key, "CS", is not found in the table, then the lookup returns **nil** and for this reason entry deletion has been provided by, for instance:

```
s.enter( "CS", addressTable , nil )
```

Another mechanism provided for use with tables is the **s.scan** command which applies a procedure to every entry in a table. The strong type checking of PS-algol restricts this procedure to be of type:

```
proc( string,pntr -> bool )
```

where the parameters make the keys and values available for manipulation within the procedure. The result of the procedure should be **true**, unless the scan must be terminated prematurely, in which case the value, **false**, should be returned. A sample scan procedure is:

```
let Double = proc( string key; pntr address -> bool )
      begin
            address( house ) := 2 * address( house )
            true
      end
```

would be applied to every address in the *addressTable* by the command:

```
let n = s.scan( addressTable, Double )
```

which not only causes havoc in the table by doubling all the house numbers, but also returns the number of addresses it has found to do this to.

Tables are important within PS-algol, because they are used as the topmost structure of the "databases" within which objects are stored in order to make them persist. The commands:

```
      let DB = create.database( "addressDB", "friend")
and   let DB = open.database( "addressDB", "friend", "write")
```

which respectively create and open the database whose name is "addressDB" and whose password is "friend", return a table Abstract Data Type which is empty when the database is created.

Usually the topmost table in a database will point to further tables. Figure 5 shows the setup for the database of bibliographic references described in [PPRR24]. In this diagram can be seen:

- the name and password of the database in a box at the top;
- the top level table shown as a line down the left hand side with horizontal lines pointing to the entries in the table;

and • the entries pointing to further tables.

The organisation of this database will be described in more detail in the following sections.

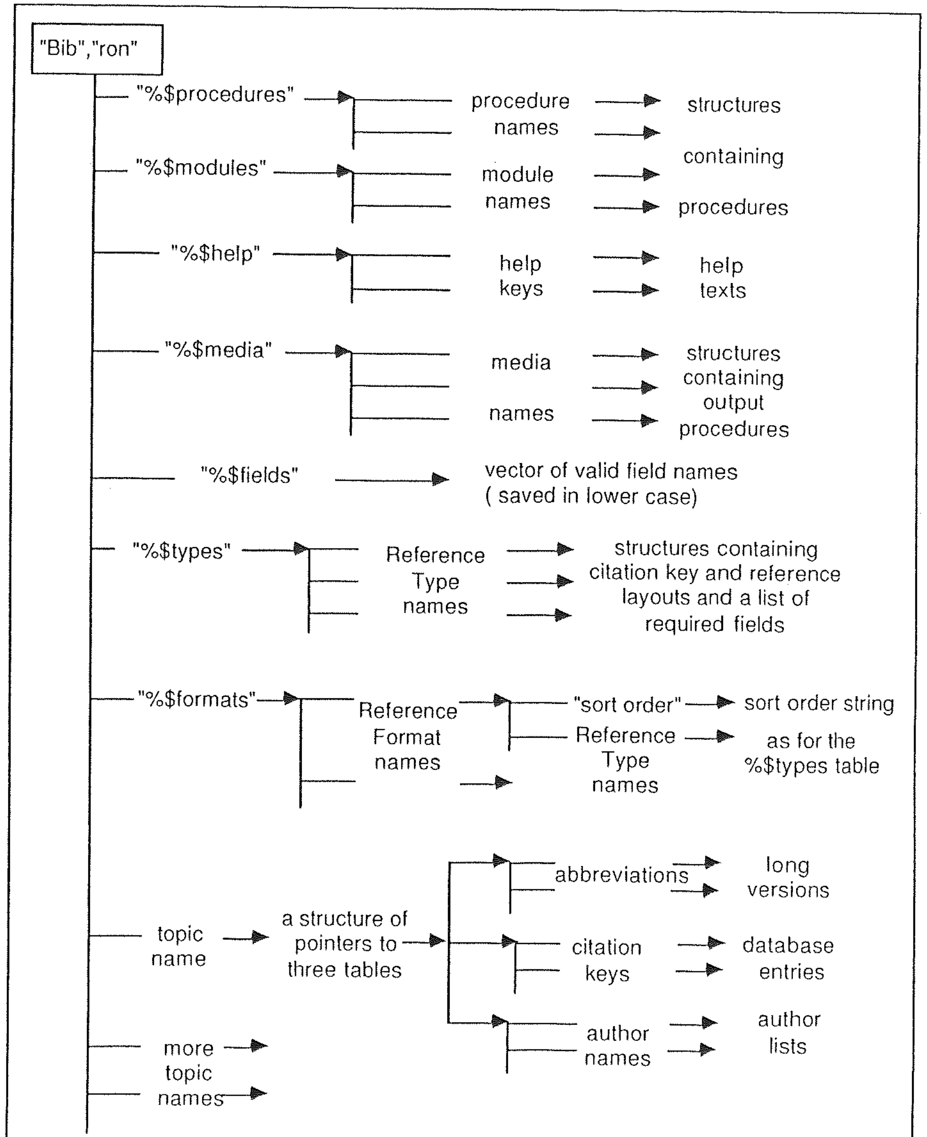


Figure 5. **The Bibliographic Database Layout.**

b) What Persists?

"Reachability" is the concept used by PS-algol to determine the persistence of a piece a data. Within a program run, data persists as long as it is reachable from some object still in scope at the current point of the program. Thus after the following:

```
structure intPack( int value )
let x := nil
        begin
                let y := 0
                let z := 1
                x := intPack(  z )
        end
```

the pointer variable *x* persists because it is still in scope. The other two variables, *y* and *z* are now out of scope, but the value of *z* has been made to persist by storing it in a structure which is pointed to by *x*, which is, as we have said, still in scope. The variable *y*, on the other hand, has no reference to it and is out of scope and so has not persisted and may be garbage collected.

The mechanism for persistence of data beyond the end of the current program run is the database, introduced in the previous section. Briefly, data will persist as long as two conditions have been met:

• it has been made reachable from the top level table of a database

and • then a **commit** command has been given.

The rules for whether a data object is reachable are:

• if it is not of type **pntr**, it must have been entered into a structure which is reachable

• a structure is reachable if it is part of a reachable super-structure

• the top-level table is reachable.

Thus in the following rather baroque structure, in which *TopLevelTable* is the top level table of some database the value of *x* will persist, via references in the objects *y*, *LowLevelTable* and *z*:

```
structure intPack( int value )
structure intermed( pntr ref , ....)
let x := 1
let y := intPack( x )
let LowLevelTable  := table()
s.enter( "x", LowLevelTable, y )
let z := intermed( LowLevelTable, ... )
s.enter( "low", TopLevelTable , z )
```

The **commit** command can be performed at any time and it has the effect of making permanent any changes made by the program to any of the databases. Thus it is

impossible to selectively abort changes which have been made. The following section deals with techniques for overcoming this by making changes as atomic transactions to the database.

c) Modelling Transactions.

Let us set up a database which will hold various collections of addresses. The program to set up the database with one collection of addresses, "MyAddresses", might look like

```
let addressDB = create.database( "Addresses", "Postman" )
let MyAddresses = table()
s.enter( MyAddresses", addressDB, MyAddresses )
if commit() = nil then write "Address Database Created o.k.'n"
                   else write "Address Database set up failed'n"
```

We now want to provide a facility for editing the database and do so as a set of editors. In top-down order these are:

• an editor for the whole database, this will operate on collections of addresses, adding new ones, deleting them and editing them;

• an editor for a single collection, which will operate on addresses in a similar manner, adding, deleting and modifying them;

• an editor for a single address, which will edit the fields of the address;

and • editors for each of the fields of an address (they call primitive editors, like the one described for strings in section 3d).

Each of these editors will be designed to operate as an all or nothing editor, returning either the old value or a new object as directed by the user. They will not make their changes to the database until the whole database is to be changed. Let us see how this can be done, by describing the superstructure of the editors from the bottom up.

The following editor edits the name of the address:

```
let EditStreet = proc( string OldStreet -> string )
        begin
                let NewStreet  = StringEditor( OldStreet )
                print "Do you want to keep the edited name?"
                if read()  = "y"     then NewStreet
                                     else OldStreet
        end
```

It is to be noted that the editor makes a new object and then returns it if the user wants to keep the changes. We have thus provided an atomic editor on the street field.

Our second editor, for a whole address, is as follows:

```
structure Address( int House; string Street, City )
let EditAddress = proc( pntr OldAddress -> pntr )
      begin
            .....                                   ! This code will contain calls
            let NewStreet := OldAddress( Street )   ! to the various field editors
            NewStreet := EditStreet ( NewStreet )   ! controlled by a form interface,
            ....                                    ! for instance
            print "Do you want to keep the edited address?"
            if read() = "y"    then Address ( ..., NewStreet , .... )
                               else OldAddress
      end
```

Here the edit transaction has been made atomic, the returned address object being either the old object or a completely new one.

The editor for "my" collection of addresses can then be written as shown in Appendix C. In this procedure, all changes are recorded in the table, *ChangesTable*, the three operations on the table operating as follows:

"add": requests a name from the user, creates an empty address object and then calls the address editor to fill in the values - the new name, value pair are then inserted into the *ChangesTable*;

"del": this calls a procedure to get the name of the address to be deleted from the user from a menu of all the names in the table. The name is then associated with the constant *del* in *ChangesTable*. Note that, we may not associate the name with **nil** as this will delete the entry from *ChangesTable*, not from the old table.

"edit": again the name of the address to be edited is requested by menu and the entry corresponding to this is looked up, first in the *ChangesTable* as this will hold the most recent value of the object, then in *OldTable*. The value is then sent to the address editor.

The names presented by *getname* are initialised to be all of the keys of *OldTable* by a standard procedure *makeKeyMenu* and is maintained by the two procedures *addKeyToMenu* and *removeKeyFromMenu*. When all of the changes have been specified, the user is asked if the changes are to be kept and if so a new table is built. This consists first of the entries of *OldTable* and then of the changes in *ChangesTable*. Otherwise, the old table is returned unchanged.

Finally, we could specify in the same way a procedure which operates on the top level table. Perhaps though a better alternative is to extend our procedure *EditAddressTable* to a more general version which edits tables whose elements are all of the same (unspecified) class. The only change required to do this is to give the procedure an extra parameter, which is a procedure which edits an element of the given type and to replace the calls to *EditAddress* by calls to the parameter, *EditElement*. The procedure declarations becomes:

```
let EditTable = proc( proc( pntr -> pntr ) EditElement; pntr OldTable ->  pntr )
```

Now we can edit my collection of addresses with:

```
MyAddresses := EditTable ( EditAddress , MyAddresses )
```

Furthermore, we can specify a version of *EditElement* which operates on tables of tables, as follows:

```
let EditTableAsElement := proc( pntr OldElement -> pntr ); nullproc
EditTableAsElement := proc( pntr OldElement -> pntr )
        EditTable( EditTableAsElement, OldElement )
```

We would like to be able to apply it to the top level table, by doing:

```
AddressDB := EditTable( EditTableAsElement, AddressDB )
```

but this would stop *AddressDB* being a 'database' (it would instead be an unattached table). We need instead to write a version of *EditTable*, which merges the changes in with the database. This is left as an exercise to the reader.

5/ Program Development and Organisation.

a) Modular Development.

Analysing a program into small manageable units is now accepted as the most efficient way of carrying out large programming tasks. In algol-like languages, the various functions of the program are carried out by procedures. In PS-algol, procedures are first-class objects and so may be manipulated in the same way as other data types, in particular they may be assigned to variables, passed as parameters to other procedures and stored in the database. One immediate gain from this is that once the program has been divided into procedures, each procedure can be put into a different source file and be compiled separately, with consequent savings in debugging time.

Such a module usually consists of three parts :

- in the first part all of the data required is retrieved from the database along with any procedures called by this one;
- then follows the body of the procedure;
- finally the procedure is wrapped up in a structure and stored in the database.

In practice, it may be better to code more than one procedure within a module, where the procedures are small or where they share a lot of data. In this case, the procedures may be packaged together into a single structure for storage in the database or they may be packaged and entered separately. The decision on which procedures to package together will be determined entirely by how meaningful such a grouping is to the programmer.

To illustrate the technique, let us look at the list handling procedures given in part 1. The source required to put the three procedures, *start, add* and *print*, into the database is as follows:

```
structure anyListElement( pntr value, next )
let anyList := nil

let startAnyList = proc(); anyList := nil

let addAnyList = proc( pntr newValue )
        anyList := anyListElement ( newValue, anyList )

let printAnyList = proc()
        begin
                let P := anyList
                while P ~= nil do
                    begin
                        let V = P( value )
                        if V is stringPack do print V( sValue )
                        if V is intPack do print V( iValue )
                        P := P( next )
                    end
        end

structure listPackage(    proc( ) StartAnyList ,
                          proc( pntr ) AddAnyList;
                          proc( ) PrintAnyList )
let theListPackage = listPackage( startAnyList, addAnyList, printAnyList )
let procdb = open.database( "Procedure Library", "friend", "write" )
s.enter( "List Package", procdb, theListPackage )
if commit() = nil do write "List package entered successfully'n"
```

The final part of the program declares a packaging structure. Note that the field names must be different from any other name in the scope - a consistent use of case is one systematic way of getting round the inconvenience of this. Then the procedures are packaged and entered into the database.

The retrieval of procedures is illustrated by the following, which retrieves the *startAnyList* procedure:

```
structure listPackage(    proc( ) StartAnyList ,
                          proc( pntr ) AddAnyList;
                          proc( ) PrintAnyList )
let procdb = open.database( "Procedure Library", "friend", "read" )
let theListPackage = s.lookup( "List Package", procdb )
let startAnyList = theListPackage( StartAnyList )
```

b) Maintaining A Procedure Library.

Given the ability to divide up the program source as described above, it becomes an attractive option to create and maintain procedure libraries systematically. Firstly, this gives us the possibility of storing support information about the procedures and, secondly, it allows us to overcome the following problem:

Suppose that we have entered what purports to be a minimum procedure into our database by running the following program:

```
structure procpac1( proc( real, real -> real ) xproc1 )
let min = proc( real a, b -> real ); if a<b then b else a
s.enter( "min", procdb, procpac1( min ) )
....... commit()......
```

and then we also add in a procedure which calculates the minimum of a vector:

```
structure procpac1( proc( real, real -> real ) xproc1 )
structure procpac2( proc( *real -> real ) xproc2 )
let min = s.lookup( "min", procdb )( xproc1 )
let minvec = proc( *real data -> real )
        begin
                let smallest := data( lwb( data ) )
                for i = lwb( data ) to upb( data ) do smallest := min( smallest, data( i ) )
                smallest
        end
s.enter( "minvec", procdb, procpac2( minvec) )
....... commit()......
```

Now we retrieve and use *minvec* and get surprisingly large numbers, due to the error in *min*. If we now, edit, recompile and re-run the program with *min* in it, we would expect the problem to go away, but it doesn't! This is because *minvec* is still pointing to the old version of *min*. It is essential to re-run (but not to re-compile) the module with *minvec* in it - this will result in *minvec* correctly pointing to the updated form of *min*. This is a general problem and it would be useful to have a mechanism whereby updating a procedure would at least alert the programmer to the fact that the modules containing procedures which call this one must be re-run. Note that this feature of the database system is not undesirable, since it permits a given user to keep on using an old version of a library procedure or to switch to a new version as required.

Let us therefore use the following standardised structure to hold the procedures:

```
structure intermediate(
                pntr procpaki;          ! points to the procedure packaged as below
                *string depended;       ! a list of the procedures which call this one
                string datestamp;       ! the time at which the proc was put into the library
                string author;          ! the author
                string description )    ! a description of the procedure
```

The *procpaki* field points to a structure which is always called *procpak* and which always has a single field, named *xproc*. This field is a procedure, the type of which varies from procedure to procedure. Appendix D contains two programs written to maintain such a library. The first program, *procdbmaker*, creates the database, while the second, *prcdblister*, lists the contents of the database in a standard layout.

The main function of *procdbmaker* is to set up an empty database to hold the procedures and to put in two procedures, one of which inserts procedures (*prcput*) and one which retrieves them (*prcget*). The program starts by checking whether the database already exists and if it does checking with the user that it is all right to delete the old one. Then the two procedures are defined and inserted into the database. The method of inserting procedures is illustrated by the lines which insert *prcput* -

```
{ structure procpak(proc(string,pntr,*string,string,string) xproc)
  prcput( "prcput",procpak( prcput ), vector 0::0 of "", "RC", "Put procedures into the database")}
```

The entry consists of a single block containing one command to specify the procedure's type and another to call *prcput* to insert itself. Note that the two lines have been put between curly brackets, which are synonymous with **begin** .. **end** in PS-algol. This has allowed us to use a uniform storage method for procedures of different types. Since all identifiers have to be unique the following:

```
structure procpak( proc( string ) xproc )
prcput( "proc1", procpak( proc1), .... )
structure procpak ( proc( int ) xproc )
prcput( "proc2", procpak( proc2), .... )
```

would be rejected by the compiler as the names *procpak* and *xproc* have been defined twice within the same block. By putting in the brackets we have thrown away all the packaging, which won't persist beyond the end of the block, and so avoided having to think up and remember new packaging names for every procedure we encounter.

The *prcput* procedure takes in five parameters:

the procedure name;
a pointer to the packaged procedure;
a vector of the names of procedures which this one calls;
a string containing the author's name;
and a description of the procedure.

It then proceeds as follows:

It checks if the procedure is already in the library - if it isn't just load it.

If it already exists, but is of the same type also proceed with loading it, but if it is of a different type, display a warning message and give the user a chance to abort the load.

Print a list of the procedures which call it and then clear the list.

Scan the list of procedures which it calls and put an entry in each of their lists of dependent procedures.

Finally commit the database entry.

prcget is much simpler, consisting just of a lookup followed by a dereference of the procedure pointer field. Note that *prcget* is not packaged in the *intermediate* structure, since it contains the code which unpackages such structures. Therefore it could not be retrieved from the database until it had already been retrieved.

prcdblister prints a table of the information about the procedures in the library in the form shown in Figure 6. It starts by pulling some utilities out of the library in order to use them itself. The technique for getting procedures out of the library is analagous to the technique for putting them in. The packaging information is thrown away immediately it has been used by surrounding the retrieval in curly brackets. The procedures retrieved are:

fillstring - which fills out a string with spaces to be of a specified length
columnator - which puts a vector of strings into columns
and *putFile* and *putLine* which were defined above.

The program then uses the **s.scan** command of PS-algol to apply the procedure *pdbscan* to every entry in the library. This procedure displays the procedure name, its type and some of the support information on one or more lines of the table. The only point of interest is that it makes use of the standard function, **class.identifier**. This returns the type of an object given a pointer to it. The class identifier of one of the procedures in the library looks like:

```
procpak(
proc(string,int) xproc)
```

This is passed to the procedure, *someof*, to reduce this to

```
(string, int )
```

the form in which it is printed.

```
-----------------------------------------------------------------------------------------------
prcget(string -> pntr)                                                                        |
prcput(string,pntr,*string,string)      Nov 6/86 14:52   Put procedures into the database     |
columnator(*string,*int,*int,*int       Dec 3/86 17:59   Prepare columns of text for display  |
          -> string)                                                                          |
fillstring(string,int -> string)        Dec 3/86 17:59   Fill out a string with spaces        |
nothing()                               Dec 3/86 17:59   Do nothing at all                    |
getfile(string)                         Dec 3/86 17:51   Make input spool from the keys or file |
getline(-> string)                      Dec 3/86 17:51   Get a string from the output spool   |
putfile(string)                         Dec 3/86 17:50   Send output spool to the screen or file |
putline(string)                         Dec 3/86 17:51   Send a string to the output spool    |
error.message(string,int,int)           Nov24/86 11:07   Display a message in a screen window |
ieditor(string,int,int,int,int,int      Nov17/86 17:09   Integer editting in a screen window  |
           -> int)                                                                            |
max(int,int -> int)                     Nov17/86 17:09   Returns larger of two integers       |
min(int,int -> int)                     Nov17/86 17:09   Returns smaller of two integers      |
seditor(string,string,int,int,int,      Nov17/86 17:09   Text editting in a screen window     |
          int -> string)                                                                      |
help(cstring,pntr)                      Nov 6/86 15:24   Helper                               |
form.generate(-> pntr)                  Dec 1/86 12:33   Generate a form ADT                  |
-----------------------------                                                                 |
16 procedures in the database                                                                 |
-----------------------------------------------------------------------------------------------
```

Figure 6. The Contents of the Procedure Library.

c) Organising The Database For A Program.

The information kept in the persistent store by a large application may be partitioned between programs and data as follows:

• programs:	• procedures representing large program modules;
	• application-specific utility procedures;
	• utilities of general applicability, which may be re-usable;
• data:	• meta-data
	• the "actual" data entered by the user.

A useful organisation of the database which incorporated all of these would to be:

- one database devoted to general utilities, with a standard password, like "friend" - it would be maintained by the mechanisms described in 5b, above;
- everything else to be stored in a database specific to the application;
- one table of this database to include the application-specific utilities - this would also be maintained by the method outlined in 5b, above;
- one table of the main program modules, so that these can be accessed separately;
- one table each for the various types of metadata held in the system.
- one structure held at the top-most level of the database which contains the current values of all the global variables in the system;
- other data to be entered in whichever structure seems most appropriate.

The database for the Bibliographic Reference Database Program, shown in Figure 5, is organised in this way. The top two entries in the table show the utilities and modules of the program. Then follows the meta-data, including help information and a description of the types of reference which the program knows about. Finally, at the bottom of the diagram, appear the user-entered data, which is grouped into "topics".

d) Organising A Program.

The organisation of the program for a given application will be largely determined by analysis into manageable units. The low-level utilities should be stored in a database as described above so that they can be retrieved and used from any higher level module. The larger modules call themselves in an order determined by which operation has to occur at each stage. They can be linked together either sequentially if a sequential dialogue with the user seems to suit the application. Alternatively, they can be linked together via the PS-algol menu construct or by a form interface such as the one which is described in section 3c, above, and 6c and 6d, below.

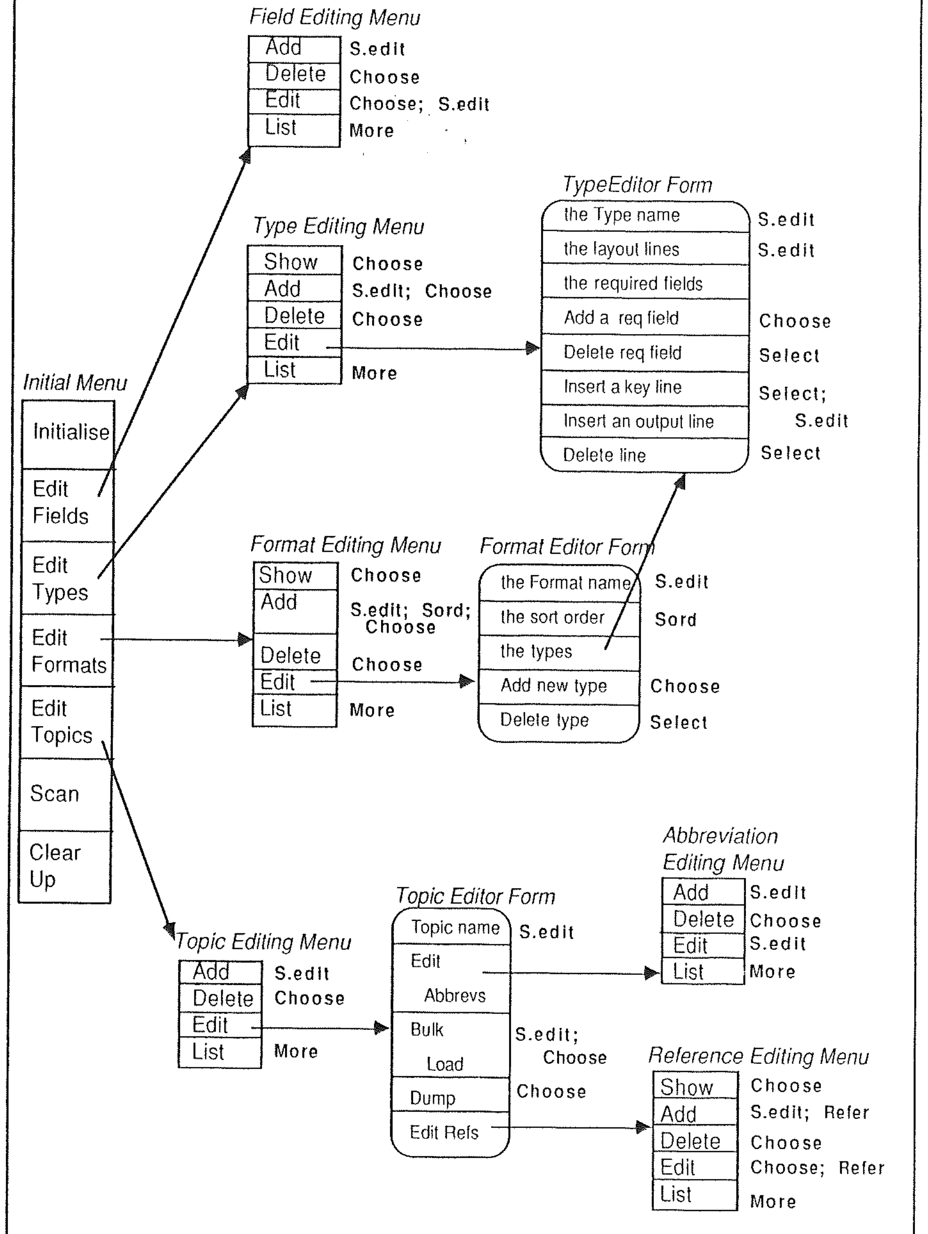


Figure 7. The Bibliographic Reference Menu Hierarchy.

Figure 7 shows the menu hierarchy within the Bibliographic Reference Database Program. Flow of control passes from left to right in the diagram via menus, shown as rectangular boxes, and forms, shown as rounded boxes. At the leaves of the hierarchy appear dialogues of user interaction. For instance, a bulk load is achieved by selecting the sequence "Edit Topics", "Edit" and "Bulk Load" and then entering into a dialogue using the String Editor, to enter the file name, and the Chooser to determine the format that the file is in.

The Chooser mentioned here is a utility which provides a method of selecting between a set of objects by menu. It is an example of the kind of flexibility that can be offered by PS-algol. In the database, there is a table of "bulk loaders", one for each format that the program knows about. When the Bulk Load option is chosen, the bulk loading module examines that table and presents the user with a choice of the loading methods available. This means that if, at some future time, a new file format is required, it is only necessary to write a procedure which loads a file of that format and enter it into the table of bulk loaders. The next run of the main program will then automatically include that option in the menu.

6/ Using First Class Procedures.

a) Procedure Variables.

Procedures are first-class objects in PS-algol. The first implication of this is that there can be variables whose type is **proc**. The type matching rules also take account of the parameter types of the procedure. Thus the two procedures:

```
let nothing = proc(); {}
```

and

```
let loseA = proc( string a ); {}
```

will not match types. Note, however, that the parameter names are ignored. Therefore,

```
let loseB = proc( string b ); {}
```

will match *loseA* as both are of type, **proc(string)**.

In all the examples of procedures so far, the procedures have been assigned to constants - as they have indeed for the three procedures just given. In order to define a variable to hold a procedure, give a declaration with a colon, as usual, eg:

```
let variableNothing := proc(); {}
```

which could at a later stage, be re-assigned to another (parameterless) procedure as required.

The use of procedure variables is mandatory in the case of recursive procedures. Thus factorial may not be defined:

```
let factorial = proc( int n -> int )
        if  n = 1    then 1
                     else n * factorial( n - 1 )
```

as this will not compile, since *factorial* is not yet defined when it is called within itself. There is no object called *factorial* until the procedure body is finished. To fix this put

```
let factorial := proc( int n -> int ); nullproc
factorial := proc( int n -> int )
        if  n = 1    then 1 else n * factorial( n - 1 )
```

where **nullproc** is a null procedure body. Application of a **nullproc** gives rise to a PS-algol event.

There are three substantial benefits that are derived from having first-class procedures within the language:

- the ability to develop a program incrementally;
- the ability to model actions;

and • the ability through procedure generating procedures to represent data by Abstract Data Types.

The former was discussed in section 4a above. We now continue by showing how actions can be modelled and how to provide ADT's.

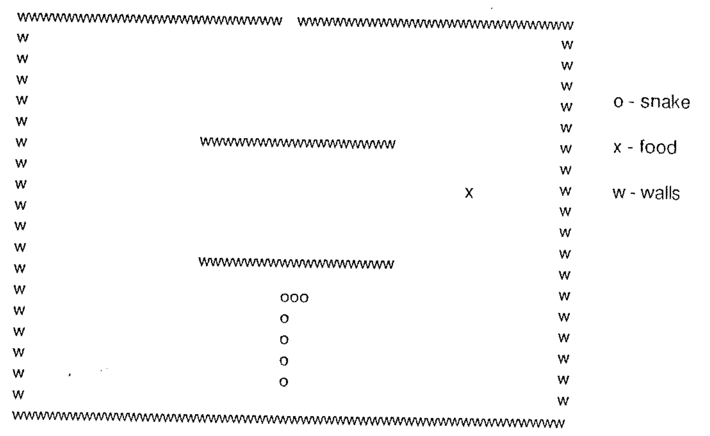


Figure 8. The Snake Display

b) Modelling Actions.

Consider a program for displaying the Snake game commonly found on micros. In this game, there is a snake, initially of length *l*, which travels either horizontally or vertically at constant speed about the screen. On the screen there are walls in the form of lines and food in the form of dots. The walls must be avoided - running into one loses a life. The food must be picked up to score points and make the snake grow. At any time, the direction of the snake may be changed by 90 degrees, by pressing one of the mouse buttons. Initially the snake is travelling upwards and so only the left and right buttons have any effect - the up and down buttons do nothing at all. Similarly when the snake is travelling left or right, only the up and down buttons will change the direction of movement. The program display is shown in Figure 8.

Perhaps the obvious outline for this program might be:

```
let direction := "U";   let crashed := false;            let maxwell := locator()
let show := proc()                 ! a procedure which moves the snake one screen unit and
      .......                      ! checks whether it has hit a wall or come across food
while ~crashed do
   begin
      if maxwell( the.buttons )(1) and ( direction = "L" or direction = "R" ) do direction := "U"
      ....  ditto for the other three directions
      case direction of
            "U":  { Y := Y + 1;  show() }
                   .. entries for other three directions
   end
```

This procedure, while being small and neat, is inefficient in that it includes a great many logical expressions, which need to be evaluated. Replacing the string variable, *direction*, with a set of boolean flags does not help a great deal. A better implementation has the following outline:

```
let crashed := false;     let maxwell := locator()
let show := proc()                ! a procedure which moves the snake one screen unit and
      .......                     ! checks whether it has hit a wall or come across food
let nothing = proc(); {}
let move := nothing
let pressUp := nothing
....  ditto to declare pressDown, pressLeft and pressRight
let moveUp = proc(); {  Y := Y + 1;  show() }
....  ditto to declare moveDown, moveLeft and moveRight
let changeUp = proc()
   begin
      move := moveUp
      pressUp := nothing;          pressDown := nothing
      pressLeft := changeLeft;     pressRight := changeRight
   end
....  ditto to declare changeDown, changeLeft and changeRight
while ~crashed do
   begin
      if maxwell( the.buttons )(1) do pressUp()
      .... for the other directions
      move()
   end
```

The main loop of the program simply circles round checking the buttons, calling the corresponding variable procedures *pressUp*, etc, when one of them is pressed. The loop ends by calling the variable procedure, *move*, which moves the head of the snake one square. The *press* procedures either do nothing, by being set to procedure, *nothing*, or call one of the *change* procedures. The *change* procedures simply set *move* to the appropriate direction and also set the values of the *press* procedures to respond appropriately to the buttons. The *move* procedures change the current position and call procedure, *show*. By simplifying the boolean expressions, the program runs faster. The use of variable procedures also seem to provide a model which is a good metaphor for what is happening on the screen. The whole program is listed as Appendix E.

c) The Form of Light Buttons Revisited.

In section 3c, we constructed a light button system, which contained a procedure for displaying boxed text and another for checking which box had been pointed at by the mouse. The monitoring procedure returned the text from the box and from this it would be possible to write programs which react to the pressing of different buttons.

We now propose to modify the system to make use of first-class procedures, so that we can (a) make the monitoring procedure activate an associated procedure when the button is pressed; and (b) give the calling procedure the ability to erase buttons from the screen if required. Let us start by extending our button structure:

```
structure button(string strip; int xo, yo, xs, ys; #pixel savedArea;
                                                       proc() action; pntr nextButton)
```

and then extend the *NewButton* procedure to take in an action procedure and to return a procedure which erases the button, viz.

```
let NewButton = proc( string message; int xorg, yorg; proc() inAction -> proc() )
     begin
          .... as for original NewButton until
          ButtonList := button ( message, xorg, yorg, xsize, ysize,
                                                       save, inAction, ButtonList )
          proc();  copy save onto box
     end
```

The procedure now returns a procedure to the calling program, which it can store and use at any time to erase the button from the screen. The *MonitorButton* now becomes:

```
let MonitorButton = proc()
     begin
          let maxwell := locator()
          let found := false
          let P := ButtonList
          while ~found do
             begin
               while ~maxwell( the.buttons )( 1 ) do maxwell := locator()
               while maxwell( the.buttons )( 1 ) do maxwell := locator()
               let X = maxwell( X.pos )
               let Y = maxwell( Y.pos )
               P := ButtonList
```

```
              while ~found and P ~= nil do
                   { if P(xo) <= X and X <= P(xo)+P(xs) and
                              P(yo) < Y and Y <= P(yo)+P(ys) do found := true
                     P := P( nextButton )                    }
              end
          let areaBox = limit screen to P(xs) by P(ys) at P(xo), P(yo)
          not areaBox onto areaBox                ! invert the light button during the
          P(action) ()                            ! procedure
          not areaBox onto areaBox
     end
```

The inner loop of the procedure cycles round until the mouse has been clicked over one of the light buttons of the form - variable *P* points to this button. As soon as a button is clicked over, its screen area is inverted and the procedure associated with it is applied.

d) Representing Data By Abstract Data Types.

The use of Abstract Data Types (ADTs) to represent data brings two main benefits through the separation of the user from the data representation. Firstly, the user need not be concerned by the representation and can concentrate on its functionality. Secondly, the data is protected from misuse. To illustrate the way in which ADTs are programmed in PS-algol, we return again to our light button example.

In essence, we have already managed to separate the data representation from the calling program, as the only interface the program has consists of the three procedures, *start*, *add* and *monitor*. However, we have ignored the following problem. Suppose we put up a light button form, in which one of the associated actions wants to set up its own form. So the action procedure calls *add* a few more times and then *monitor*. Now the program is monitoring both sets of light buttons, when we only want it to see the second set. What we actually want is a separate list for each form and this may be done by providing a form generator, which returns a new form as a set of procedures. This will set up "form" as an ADT, with two operations: add a new button and monitor the form. The code for this is:

```
structure button(string strip; int xo, yo, xs, ys; #pixel savedArea;
                                              proc() action; pntr nextButton)
let FormGenerate = proc( -> pntr )
  begin
     let ButtonList:= nil
     let NewButton = proc( string message; int xorg, yorg; proc() inAction -> proc() )
          begin
               .... as for NewButton in section 6c
          end
     let MonitorButton = proc()
          begin
               ....as for MonitorButton in section 6c
          end

     structure FormPack (   proc( string, int, int, proc() -> proc() ) newbutton;
                            proc( ) monitorbutton )
     FormPack ( NewButton, MonitorButton )
  end
```

Now, every time *FormGenerate* is called, a new *ButtonList* is created, associated with its own instantiation of the two procedures. Thus the *monitor* procedure will only monitor those buttons put up by the associated *NewButton* procedure. We can then lay out a program with two levels of form as follows:

```
structure FormPack (    proc( string, int, int, proc() -> proc() ) newbutton;
                        proc( ) monitorbutton )
let aButtonProc = proc()
     begin
          let secondForm := FormGenerate()
          let secondNew := secondForm( newbutton )
          let secondMonitor := secondForm( monitorbutton )
          ......
          let secondEraser = secondNew( .......)      ! various buttons added
          ......
          secondMonitor( )
          secondEraser()                              ! remove light buttons before quitting
          ......
     end

let  firstForm := FormGenerate()
let firstNew := first.form( newbutton )
let firstMonitor := firstForm( monitorbutton )
....
let firstEraser := firstNew( "Call second form", 100, 100, aButtonProc )
....
firstMonitor()
firstEraser()
```

In this program, the first form is put up and when the light button, which lis abelled "Call second form" is clicked over, the procedure, *aButtonProc*, is called. This puts up the second form and calls *secondMonitor*. At this point, only the light buttons of the second form are responsive. When *secondMonitor* is finished, then the light button is cleared away.

That was our last look at the interactive form program. As outlined here, only two operations have been provided. Appendix F shows a generator for a fuller ADT for forms. This includes the following operations:

FormShow: redisplays a light button given a pointer to it.

FormAdd: takes in a string to be displayed, box position and dimensions and an action procedure and returns a pointer to the list element.

FormClear: clear the form from the screen.

FormRemove: remove a button from the form.

FormUpdate: update the text seen in a light button.

FormMouse: return a pointer to the button clicked over.

FormMonitor: keep monitoring the form, applying the action procedures of any light buttons clicked over, until the light button associated with the provided *fender* procedure is selected.

fender: this is not an operation of the ADT, but an action procedure provided as part of the package. One of the calls to *FormAdd* must associate a light button (presumably marked "quit") with the procedure so that *FormMonitor* may terminate.

7/ Interactive Compilation.

a) Introduction.

PS-algol programs are strictly type checked and this makes the writing of polymorphic procedures impossible by normal programming methods. However, there are two standard procedures provided which allow the programmer to get round this:

class.identifier which takes in a pointer, *P*, and returns a string which contains the structure class name and a list of the names and types of the fields of the structure of *P*;

and *compiler* which takes the name of a file containing the source code of a procedure and a pointer to a packaging structure for the compiled procedure and procedure compiled and packaged.

Using these two procedures, it is possible to create effectively polymorphic procedures, which are given their input in the form of a pointer to a structure of an unknown type and then proceed as follows:

Get the class identifier.

Build a procedure in a string variable from information found in the class identifier, which will handle objects of this structure.

Output this procedure to a file.

Call the compiler to compile this.

Run it against the input object.

b) Calling The Compiler.

The compiler procedure has the following form:

compiler = **proc**(**cfile** *sourceFile*; **cpntr** *holder* -> **pntr**)

where *sourceFile* is the name of a file containing the source of a PS-algol procedure and *holder* is a pointer to a structure which will hold the compiled form. The procedure returns a pointer to this structure with the compiled form inserted.

For example, if the file named *exampleProc* contains a proc whose declaration is:

```
proc( string inString -> int )
```

the compiled form would be generated by:

```
structure procHolder ( proc(string -> int) theProc )
let Package = procHolder( proc( string inString -> int ); nullproc )
let compiledPack = compiler( exampleProc, Package )
let compiledProc = compiledPack ( theProc )
```

In this code, *Package* is the empty structure sent to the compiler (it needs this as it cannot create a structure to return the procedure in, only fill an empty one); *compiledPack* is the procedure compiled, still within its packaging, and *compiledProc* is the unpackaged procedure which may be applied, as in:

```
compiledProc( "ABC" )
```

or stored or used in the same way any other procedure might be used.

It is unfortunate that the chosen interface for the compiler is the file, but Appendix G gives a set of procedures, which are useful interfaces to the run-time compiler and to the *class.identifier* function. The first procedure is:

compile = **proc**(**cstring** *source*; **cpntr** *holder* -> **pntr**)

which replaces the **file** parameter with a string parameter. *compile* operates by creating a temporary file in /tmp and submitting it to *compiler*.

Five other procedures are given in Appendix G. These are a procedure, *reportErrors*, which unpacks and displays error messages produced by the compiler and four string handling procedures which take in the class identifier and produce the following:

CltoStruct - produces, as a string, a PS-algol structure declaration.

CltoFields - produces, as a vector of strings, the set of field names.

CltoSname - produces, as a string, the structure name.

CltoTypes - produces a parsing of the field types as a vector of linked lists of the following class of objects:
structure *typeList*(**string** *S*, *R*; **pntr** *N*)
where "c"s and "*"s are separated from the rest of the type information, so that for instance *cint is represented as
("*", "cint",)-> ("c", "int,)-> ("int", "", **nil**)
that is the fields are current String, Rest of string and Next element.

c) Producing Fully Polymorphic Procedures.

One reason for compiling procedures during the run of a program is to provide procedures which will deal with the structures which have yet to be specified. Given such a structure, it is possible to discover the structure name and the names and the types of the fields by use of *class.identifier*. It is not, however, possible to access the values of the fields directly - there is no way of writing the value of a field whose name I don't know when I'm writing the program. Our first example shows how essential this information is.

We require a procedure which given an arbitrary record will report whether or not a particular string is present as the value of one of the string fields. We will assume that the record has a flat structure. That is we will not chase down any pointer fields to look at sub-structures - to do so would require a procedure which recursively recompiles a sub-procedure each time it goes down a level.

Our procedure, *checkRecord*, makes use of the persistent store to avoid unnecessary recompilation (the database browser uses the same idea). A table, called *ourTable*, is set up to contain all the sub-procedures that have so far been encountered. There will be one such sub-procedure for each structure class that is passed to *checkRecord* and each new one is placed into *ourTable* keyed by the class identifier. When it is called, *checkRecord* first consults this table to see if the structure has already been encountered. If it has, the sub-procedure is merely unpacked from the table and used. Otherwise, a new sub-procedure must be constructed, compiled and stored in the table, before it is used.

With these introductions aside, the procedure is presented as Appendix H. The code has been annotated with the following code letters:

A - the table look up section described above

B - a preliminary check is made to see if there are any string fields and if not, a standard procedure which reports the fact is created. Note that this does not need to be compiled dynamically as it is completely known when the program was written.

C - the sub-procedure is built up in the string, *source*, by calls to the procedure, *outLine*, entirely from string constants and the class identifier. For the structure:

aRecord.Sruc(**string** *A*; **bool** *B*; **cstring** *C*; **pntr** *D*)

the following would be built:

```
proc( pntr RR; string SIN -> bool )
  begin
    structure aRecord.Sruc( string A; bool B; cstring C; pntr D )
    let result := false
    if RR( A ) = SIN do result := true
    if RR( C ) = SIN do result := true
    result
  end
```

In this, the parts of the procedure which are built from the class identifier are emboldened - the rest is statically determinable. Note that in the listing of the program, the statically determinable part of the constructed procedure are printed in a different font.

D - the sub-procedure is compiled. If there are any errors, the procedure, *reportErrors* is called to print them out.

E - finally the compiled procedure is run using the input parameters of *checkRecord* as its arguments and returning its output as the result of *checkRecord.*

d) An Object Copy Procedure.

Note: this section should carry a Government Health Warning. It includes the technique of compiling and executing procedures which produce strings from which a larger procedure is constructed, compiled and executed. This is clearly a complex process and its description is necessarily complex to match. The example is included to round off this tutorial in a way which illustrates the power of the language.

In an earlier section, the problem of making a new copy of a complex object was mentioned. It will be recalled that the assignment of a pointer type variable:

A := *B*

merely makes *A* point to the same object as *B*, so that changing a field of *A* also changes *B*. To actually make *A* be a new object it is necessary to do the following in the case where *B* has the structure:

copiee(**string** *a*; **int** *b*; **real** *c*)

we require:

A := *copiee*(*B(a)*, *B(b)*, *B(c)*).

To devise a general procedure which will allow us to write -

A := *objectCopy*(*B*)

is a non-trivial problem. Were we only dealing with flat objects, our job would be comparable with the *checkRecord* example just discussed, but in dealing with any object the procedure must recursively recompile code as it goes down pointer and vector levels. One solution to the problem is given as Appendix I.

The outline of the procedure is much the same as for *checkRecord*. The technique of storing procedures as new structures are encountered has been omitted for brevity. A sub-procedure is built into the string, *source*, by calls to the procedure, *outLine*. The sub-procedure is then compiled before being run with the same parameters as *objectCopy* itself. The lines commented with an "A" are the ones which accomplish this. However, the building of the sub-procedure is a far more complex task.

The procedure declaration is put into *source* at the top of *objectCopy* and the rest of the code is built as **begin** - **end** blocks by recursive calls of *makeCode*. The kind of block constructed by *makeCode* is illustrated by the following which would be generated by structure, *copiee*, above:

```
begin
        structure copiee( string a; int b; real c)
        let X1 := A(a)
        let X2 := A(b)
        let X3 := A(c)
        copiee( X1, X2, X3 )
end
```

The blocks are built up in the string variable, *tSource*, by a set of procedures like *tLine*, which attempt to make the block indent nicely. This is accomplished in the lines commented with "B". *makeCode* takes as parameters a pointer to the object for which a block must be built and a string which will be its name within the sub-procedure being built. For the top level object this will be "A". If the object had a pointer field named *P*, then when that field was processed, it would have the name "A(P)". The block begins with the **begin** and the structure declaration and then puts in one assignment for each field of the structure. The right hand sides of these assignments are built up by the *parse* procedure, whose body consists solely of a **case** statement which makes use of the *typeList* structure described above to determine the nature of the expression, as follows:

If the first part of the type is a scalar, the expression is simply a dereference of the field. The expression to do this is held in the variable, *FnameF*. It consists of the name of the object as seen from the top level object followed by the field name in brackets. Continuing our example, if *A(P)* has integer field *I*, put in "A(P)(I)".

If the first part is a "c" for constant, ignore it and recall *parse* on the rest of the type information.

When the field is a pointer field, deal with it by building another procedure in *pSource,* which recursively calls *makeCode* on the sub-object. Note that it won't do to just call *makeCode* recursively as it is not possible to name the sub-object statically. Therefore a little procedure must be built which takes in the whole object being processed and an incarnation of *makeCode* and produces as its result the block which creates the copy of this field. For the top-level structure,

```
copy2( string z; pntr r)
```

such a procedure to build the block for field, *r*, would be:

```
proc( proc( string, pntr -> string ) MAKECODE; pntr PP -> string )
   begin
        structure copy2( string z; pntr r)
        MAKECODE( "A(r)", PP(r) )
   end
```

The result of such a procedure, which will be a block, can then be embedded into *tSource* in exactly the same way that the field identifier was for scalars.

The procedure for dealing with vectors is similar if a little more complicated. so we will deal with it in more detail.

Let us consider an example of a structure with two vector fields:

```
copy3( *cpntr v1; **int v2 )
```

The first decision to be taken is which of the two PS-algol constructors of vectors will be used for the vector expression. At first sight, we might choose the "**vector**" constructor since it can be generated statically - and this remains true even for multi-dimensional arrays. However we will choose the "@" notation because the **vector** notation will not do for vectors of constants. If an attempt is made to insert :

```
begin
        let X := vector lwb( A(v1) ) :: upb ( A(vp1 ) of nil
        for i = lwb( A(v1) ) to upb ( A(v1) ) do X(i) := A(v1)(i)
        X
end
```

into the *v1* field then a type mismatch will occur, since *X* is of type ***pntr**.

Having decided upon the @-notation, it is necessary to write a procedure to parse a vector object and produce something of the form:

```
@ lwb( A(v1)) of cpntr [ ..... ]
```

This is the procedure *vParse*. It calls itself recursively to deal with vectors of vectors and it calls *makeCode* to deal with vectors of pointers. It builds its output in three parts: *vStart* which contains the "@ lwb(X) of cpntr [" part; *vEnd* which consists of just the "]"; and *vBody*, which contains the element list. *vStart and vEnd* can be constructed statically, but *vBody* can only be generated by compiling a procedure and running it as there is no way of knowing how many elements there are in the list. The string, *vStart* is constructed from the two input parameters of *vParse* - *VS*, which contains the name of the vector as seen from the topmost object, and *VT*, which points to a *typeList* structure for the vector. The argument of **lwb** is *VS* while the type declaration is *T(R)* - this is the reason for the existence of the *R* field of *typeList.*

The rest of *vParse* is concerned with the construction, compilation and execution of a procedure of the form:

```
proc( pntr VP, VTT; string VF; proc( string, pntr -> string )
                                   MAKECODE, VPARSE -> string )
     begin
          structure copy3( *cpntr v1; **int vvi )
          let vOut := ""
          for i = lwb( VP( v1 ) ) to upb( VP( v1 ) ) do
             vOut := vOut ++ MAKECODE( VF ++ "(i)", VP( v1 )(i) ) ++ ",'n"
          vOut( 1 | length( vOut ) - 2 )
     end
```

in the string variable, *vSource*. When this procedure is called, it produces the list of elements, separated by commas - in this instance, these will consist of the code for the ith object, constructed with the appropriate name, supplied as the *VF*parameter indexed by "(i)". The object itself is obtained from the input parameter, *VP*, indexed by the field name and by *i*. The *makeCode* procedure is passed in in exactly the same way as for the procedures constructed by *makeCode*, itself. Similarly if this were a vector of vectors, the line which specifies an element of the list would contain a call to *vParse* via the input parameter, *VPARSE*.

The procedure, *makeUnit*, places the code to produce one element into this procedure. This varies depending upon the type of the vector, as follows:

if it is a vector of scalars, just put in *VF ++ "(i)"*,

if it is a vector of constants, recall *makeUnit* with the next part of the type information as the input parameter

if it is a vector of vectors, put in *VPARSE(VF++ "(i)", VTT) ++",'n"*

if it is a vector of pointers put in *MAKECODE(VF ++ "(i)", VP(.....) (i) ++ ",'n"*

When the *vSource* string is complete, it is compiled and run, producing a string which is embedded into *tSource* as the right-hand side of one of the " let Xi = .. " lines.

Eventually, the call of *makeCode* on the top level object finishes and the block constructed in *tSource* is returned. This is put into *source* and then *source* is compiled and run against the input object. The result of the procedure in *source* is the result of *objectCopy* itself. For further details of the operation of this procedure, a sample application of the procedure to copy the object, *OBJECT*, given by:

```
structure Main( string Mstring; int Mint; *int Mvint; pntr Mpntr )
structure Subsid( string Sstring; *bool Svbool )
let OBJECT := Main( "Main", 1, @ 1 of int [ 1, 2, 3 ],
                        Subsid( "Subsid", @ 1 of bool [ true, false ] ) )
```

is given as Appendix J.

Appendices.

Appendix A. A Simple File I/o Interface.

i) Output.

```
let theOfile := vector 0 :: 0 of ""
let numOlines := 0

let putStart := proc()                          ! initiate file output
        {       theOfile := vector 0 :: 0 of ""; numOlines := 0    }

let putLine := proc( string inline )            ! send a string to the output
        begin
                numOlines := numOlines + 1
                if numOlines > upb( theOfile ) do
                   {    let tfile = vector 1 :: upb( theOfile ) + 500
                        for i = 1 to upb( theOfile ) do tfile( i ) := theOfile ( i )
                        theOfile := tfile                       }
                theOfile ( numOlines ) := inline
        end

let putFile := proc( string fname )             ! terminate file output
        begin
                let fd = create( fname, 511 )
                for i = 1 to numOlines do output fd, theOfile ( i )
                close( fd )
        end
```

ii) Input.

```
let thelfile := vector 0 :: 0 of ""
let numllines := 0
let iPtr := 0

let getFile = proc( string fname )              ! start file input
        begin
                let fd = open( fname, 0 )
                while ~ eol( fd ) do
                   begin
                        let numllines := numllines + 1
                        if no.ilines > upb( thelfile ) do
                           {    let tfile = vector 1 :: upb( thelfile ) + 500
                                for i = 1 to upb( thelfile ) do tfile( i ) := thelfile ( i )
                                thelfile := tfile                       }
                        thelfile ( numllines ) := read.a.line( fd )
                   end
                iPtr := 0
        end

let getLine = proc( -> string )                 ! get a line of input
        {       iPtr:= iPtr + 1;
                if iPtr > numllines then "" else thelfile ( iPtr )        }
```

Appendix B: A Simple String Editor.

```
let StringEditor = proc( string title, tex; int xo, yo, xh, yh -> string )
  begin
    let editBox = limit screen to xh by yh at xo, yo
    let editSave = image xh by yh of off
    let editInner = limit editBox to xh-4 by yh - 15 at 2, 2
    copy editBox onto editSave
    xnor editBox onto editBox
    let titleBox = limit editBox to xh-4 by 10  at 2,yh - 12
    xor titleBox onto titleBox
    print title at xo + 5, yo + yh  - 11

    let leftText := "";        let rightText := ""
    let CursorImage = image 2 by 10 of on;                    let CursorPosn := 0

    let CursorDisplay = proc( -> #pixel )
            limit editBox at 5 + chWidth * CursorPosn, 15

    let showText = proc( int newCP; string newLT, newRT )
        begin
            CursorPosn := newCP
            leftText := newLT;           rightText := newRT
            xor editInner onto editInner
            print leftText ++ RightText at xo + 5, yo + 5
            copy CursorImage onto CursorDisplay ()
        end

    showText( length( tex ), tex, "" )     ! show cursor at right of text
    let exit := false
    while ~exit do
        begin
            let waiting := true;          let c := 0;                   let maxwell := locator( )
            while waiting do
                { if input.pending() do { c := read.byte(); waiting := false }
                  if waiting do {     maxwell := locator( )
                                      if maxwell( the.buttons ) ( 1 ) do { c:= -1; waiting := false } } }

            let mx := maxwell( X.pos );          let my := maxwell( Y.pos )
            case c of
              -1 : if xo <= mx and mx <= xo+xh and yo <= my and my <= yo+yh do
                     begin
                         let pos :=  ( mx - xo ) div chWidth
                         if pos > length( theText ) do pos := length( theText  )
                         if CursorPosn ~= pos do
                              { xor CursorImage onto CursorDisplay ( )
                                let wholeText =  leftText++rightText
                                let Leng = length(wholeText  )
                                showText( pos,
                                          if pos = 0 then "" else wholeText ( 1 | pos ),
                                          if pos = Leng then "" else wholeText ( pos - 1 | Leng -pos )  )     }
                     end

              10: exit := true         ! return pressed

              21:showText( 0, "", rightText )                    ! oops pressed

              127: if CursorPosn > 0 do
                      showText( CursorPosn - 1,
                                if CursorPosn = 1 then "" else theText ( 1 | CursorPosn - 1 ),
                                rightText )


              1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 22, 23, 24,
              25, 26,27, 28, 29, 30, 31:  { }                                  ! controls chars - do nothing

              default: showText( CursorPosn  +1, leftText ++ code( c ), rightText )

            end

        copy editSave onto editBox
        leftText  ++ rightText
    end
```

Appendix C. A Transaction Based Editor for a Table of Addresses.

```
let EditAddressTable = proc( pntr OldTable -> pntr )
 begin
    structure StringPack( string Svalue )
    let del = StringPack( "deleted" )
    let ChangesTable = table()
    let getname = makeKeyMenu( OldTable )
    let Operation := ""
    let NewName := "";   let NewValue := nil
    repeat Operation := getop()                          ! get operation from user
    while Operation ~= "finished" do
      begin
        case Operation of
                   "add": {  NewName := StringEditor( "" )
                             addToKeyMenu( NewName )
                             NewValue := Address( 0, "", "" )
                             NewValue := EditAddress( NewValue )  }

                   "del": {  NewName := getname()              ! from a menu of the keys of the table
                             removeFromKeyMenu( NewName )
                             NewValue := del  }

                   "edit": { NewName :=  getname()
                             NewValue := s.lookup( NewName, ChangesTable )
                             if NewValue = nil do
                                  NewValue := s.lookup( NewName, OldTable )
                             NewValue := EditAddress( NewValue )  }

                   default: {  }

        s.enter( NewName, ChangesTable, NewValue )
      end

    print "Do you want to keep the changes in the table?"
    if code( read.byte() ) ( 1 | 1 ) ~= "y" then OldTable  else
      begin
        let NewTable = table()
        let ChangeScan = proc( string name; pntr value -> bool )
          begin
            if    value = del   then s.enter( name, NewTable, nil )
                                else s.enter( name, NewTable, value )
              true
          end
        let o = s.scan( OldTable, ChangeScan )
        let n = s.scan( ChangesTable, ChangeScan )
        print n, "changes made to the table'n"
        NewTable
     end

 end
```

Appendix D. Procedure Library Maintainance Programs.

i) A Procedure Library Creator.

```
let procsdb:=open.database("utilities","friend","write")
if procsdb isnt error.record do { write "The database already exists'n"; abort}
procsdb:=create.database("utilities",,"friend")
structure intermed(pntr procpaki; *string depended;  string datestamp, descriptor)

!    add a string to a vector
let addsvec=proc(*string oldvec;string newelt -> *string)
     begin
          let ne=upb( oldvec )
          let newvec:=vector 1::ne+1 of ""
          if ne>0 do for i=1 to ne do newvec(i+1):=oldvec( i )
          newvec(1):=newelt
          newvec
     end

!    the put procedure
let prcput=proc(string procname; pntr procpntr; *string dependson; string author, desc)
     begin
          let initdeps:=vector 0::0 of ""
          if procname="prcput" do initdeps:=vector 1::1 of "everything"
          let okmess:=""
          let isithere=s.lookup( procname, procsdb )
          if isithere=nil then    ! new procedure - just enter it
               begin
                    s.enter( procname, procsdb, intermed(procpntr, initdeps,date(), author, desc) )
                    okmess:=" created"
               end
                         else     ! old proc - check type and dependents
               begin
                    if class.identifier(isithere(procpaki) )~= class.identifier( procpntr ) do
                    begin          ! new proc is of changed type
                         write "warning - type mismatch on changing ",procname,"'n"
                         write "do you want to do it? (y/n): "
                         if read.a.line()(1|1)~="y" do {write "ok - finishing'n";  abort}
                    end
                    let thedeps=isithere( depended )
                    s.enter( procname, procsdb, intermed( procpntr, thedeps,date(), author, desc ) )
                    okmess := " updated"
                    if upb( thedeps )>0 do for i=1 to upb( thedeps ) do
                         write "now update ", thedeps(i), " as well so that it uses this version'n"
               end
          if upb( dependson )>0 do for i=1 to upb( dependson ) do
               begin          ! mark procs this one depends on
                    let depper = s.lookup( dependson(i), procsdb )
                    if depper~=nil do
                    begin
                         let already:=false
                         let theothers=depper( depended )
                         if upb( theothers )>0 do for j=1 to upb( theothers ) do
                                   if procname=theothers(j) do already := true
                         if ~already do depper( depended ):=addsvec( theothers, procname )
                    end
               end
          if commit()=nil  then write procname,okmess," o.k.'n"
                                else { write "commit failed'n"; abort }
     end


!    The get procedure.
let prcget=proc(string procname -> pntr)
     begin
          let got=s.lookup( procname, procsdb )
          if got=nil do {write "procedure ",procname," not found in the database'n"; abort}
          got( procaki )
     end



!    Store put and get and commit.
{    structure procpak(proc(string -> pntr) xproc)
     s.enter( "prcget", procsdb, procpak( prcget ) ) }
{    structure procpak(proc(string,pntr,*string,string,string) xproc)
     prcput( "prcput",procpak( prcput ), vector 0::0 of "", "RC", "Put procedures into the database")}

if commit()=nil   then write "new database created'n"
                  else {write "creation fails - aborted'n"; abort}

?
```

ii) Procedure Library Lister.

```
let procsdb:=open.database("rutilities","friend","read")
if procsdb is error.record do {write "No utilities database - do pdbmaker first'n"; abort}
let prcget=
     begin
          structure procpak(proc(string -> pntr) xproc)
          s.lookup("prcget",procsdb)(xproc)
     end
structure intermed(pntr procpaki; *string depended; string datestamp, descriptor)

let fillstring={ structure procpak(proc(string,int -> string) xproc);     prcget("fillstring")(xproc)}
let columnator={  structure procpak(proc(*string,*int,*int,*int  -> string) xproc)
                        prcget("columnator")(xproc)}
let putfile={  structure procpak(proc(string) xproc);    prcget("putfile")(xproc)}
let putline={  structure procpak(proc(string) xproc);    prcget("putline")(xproc)}

!    transform the class identifier
let someof=proc(string inp -> string)
     begin
          let out:="()"
          if length(inp )> 23 do out:="(" ++ inp(15 | length( inp )-24) ++ ")"
          out
     end

!    transform the date
let dsomeof = proc( string inp -> string );    inp( 5| 3 ) ++ inp( 9 | 2 ) ++ inp( 23 | 2 ) ++ inp( 11 | 6 )

!    display details of a single procedure
let pdbscan=proc(string procname; pntr interproc -> bool)
     begin
          let CI = class.identifier( interproc ( procpaki ) )
          if procname = "prcget"      then
               putline( fillstring( "prcget(string -> pntr )", 99 ) ++ "|'n" )
                                             else
               putline( columnator( @ 1 of string [ procname ++ someof( CI ),
                                             dsomeof( interproc( datestamp ) ),
                                             interproc( descriptor ), interproc( author ) ],
                                        @ 1 of int [ 35, 14, 10, 32 ],
                                        @ 1 of int [  12, 5, 6, 6 ],
                                        @ 1 of int [ 3, 3, 2, 2 ] ) )
          true
     end


putline( fillstring( "Procedure", 40 ) ++ fillstring( "Date", 20 ) ++ "Description'n" )
putline( "-----------------------------------------------------------------------------------------'n" )
let n:=s.scan( procsdb, pdbscan )

putline( fillstring( "----------------------------", 99 ) ++ "|'n" )
putline( fillstring( iformat(n) ++ " procedures in the database", 99 ) ++ "|'n" )
putline( "-----------------------------------------------------------------------------------------'n" )

putfile( "/tmp/proclist" )
putfile( "screen" )
?
```

Appendix E. A Snake Program.

```
let cPosX := 0;   let cPosY  := 0          ! the current position
let headX= 50;    let headY= 20            ! the initial head position
let startX= 50;   let startY= 10           ! the initial tail position
let tailX:= 0;    let tailY:= 0            ! the current tail position
let trailHead := 0; let trailTail := 0     ! the index of the head and tail positions
let Xmax := 100;  let Ymax := 100          ! the dimensions of the grid
let growth := 0                            ! the growth variable
let  screen = vector 1::100 of vector 1::100 of 0
for i = 1 to 100 do screen(i) := vector 1::100 of 0
let walls = @ 1 of *pntr [ @ 1 of pntr [ point.strc( .. ), .... ] .... ]

let pressUP := proc(); nullproc            ! also declare press.down, press.left and press.right
let changeUP := proc(); nullproc           ! also declare change.down, change.left and change.right
let move := proc(); nullproc

let makeFood = proc()
    begin
        repeat
            {     seed := random( seed ); foodX:= seed rem 100 + 1
                  seed := random( seed ); foodY := seed rem 100 + 1   }
        while the.screen( foodX, foodY ) ~= 0
        print "x" at 5* cPosY, 5* foodY in smallFont using xor    ! display food
    end

let show = proc()
    begin
        crashed := theScreen( cPosX, cPosY ) ~= 0
        if ~crashed do
            begin
                trailHead := trailHead + 1; if trailHead = 1001 do trailHead := 1
                theScreen( cPosX, cPosY ) := trailHead
                print "o" at 5*cPosX, 5* cPosY in smallFont using xor       ! display new head
                if growth > 0 then growth := growth -1                   ! don't erase tail in growth period
                    else  begin
                            print "o" at 5*tailX, 5*tailY in smallFont using xor       ! erase tail
                            theScreen ( tailX, tailY ) := 0
                            trailTail:= trailTail + 1; if trailTail = 1001 do trailTail:= 1
                            if theScreen( tailX + 1, tailY ) = trailTail do tailX := tailX  + 1
                            .... ditto for tailX  -1, tailY; tailX, tailXY+1; tailX, tailY -1.
                        end
                    if cPosX = foodX and cPosY = foodY do
                        begin
                            theScore := theScore + 1
                            print theScore at statScore, statY in largeFont
                            print "x" at 5* foodX, 5* foodY  in smallFont using xor    ! erase food
                            makeFood()
                            growth := 10                               ! grow the snake length by 10.
                        end
                end
        end
```

```
! the procedures which actually move the snake
!    change the current position and call the display procedure
let moveUP = proc()
    begin
        cPosY := cPosY + 1
        show()
    end
....ditto for moveDOWN, moveLEFT and moveRIGHT

! the procedures called when a button is pressed
!    make the movement upward and the button responses set so that up and down are not responded
! to, while left and right are.
let changeUP := proc()
    begin
        move := moveUP
        pressUP := nothing;            pressDOWN  := nothing
        pressLEFT := changeLEFT;       pressRIGHT := changeRIGHT
    end
.....ditto for changeDOWN, changeLEFT and changeRIGHT

! The Main Program.
move := moveUP
pressUP:= nothing;                 pressDOWN := nothing
pressLEFT := changeLEFT;           pressRIGHT := changeRIGHT

let inner := limit screen to X.dim( screen - 4 ) by Y.dim( screen ) - 4 at 2,2
xnor  screen  onto  screen
xor inner onto inner
for x = 1 to X.max do for y = 1 to Y.max do theScreen := 0
for w = 1 to upb( walls ) do for i = 1 to upb( w ) do
    begin
        cPosX := walls( w )( i )(x.pos )
        cPosY := walls( w )( i )(y.pos )
        print "w" at 5* cPosX, 5* cPosY  in smallFONT
        theScreen ( cPosX, cPosY ) := -1
    end

cPosX := headX;  cPosY := headY
tailX := startX;       tailY := startY;       trailHead := 0
for y = startY to headY do
        {     trailHead:= trailHead + 1
              theScreen ( cPosX, y  ) := trailHead
              print "o" at 5* cPosX, 5*y in smallFONT                    }
makeFood()
theScore := 0

crashed := false
let maxwell := locator()
while ~crashed do
    begin              ! MAIN LOOP
            maxwell := locator()
            if maxwell( the.buttons )( 1 ) do pressUP()
            if maxwell( the.buttons )( 2 ) do pressLEFT()
            if maxwell( the.buttons )( 3 ) do pressRIGHT()
            if maxwell( the.buttons )( 4 ) do pressDOWN()
            move()
    end
?
```

Appendix F. An Abstract Data Type Generator for a Forms Interface.

```
structure button( string strip; int xo, yo, xh, yh; #pixel savedArea; proc() action; pntr nextButton )
let formGenerate = proc( -> pntr )
  begin
    let buttonList :=  nil                    ! the actual data structure used by this instance of the ADT, "form".

    let formShow = proc( pntr but )      ! display a light button
        begin
            let box = limit screen to but (xh) by but (yh) at but (xo), but (yo)
            let save = image but (xh) by but( yh ) of off
            copy box onto save
            if X.dim( but( savedArea ) ) = 1 do but( savedArea ) := save
            let inner = limit box to but (xh) -2 by but (yh) -2 at 1,1
            xnor box onto box
            xor inner onto inner
            print but( strip ) at but( xo ) + 5, but( yo ) + 5
        end

                                          ! add a light button to a form
    let formAdd = proc( string s; int XO, YO, XH, YH; proc() anAction  -> pntr )
        begin
            buttonList := button( s,  XO, YO, XH, YH;  image 1 by 1 of off, anAction, buttonList )
             formShow( buttonList )
            proc( );  copy buttonList( savedArea ) onto
                limit screen to buttonList (xh) by buttonList (yh) at buttonList (xo), buttonList (yo)
        end

    let formRemove = proc( pntr but )  ! remove a light button from the form and the display
        begin
            let drop := buttonList;let lag := nil
            while drop ~= but  and drop ~= nil do {  lag := drop; drop := drop( nextButton ) }
            copy drop( savedArea ) onto limit screen at drop( xo ), drop( yo )
            if lag = nil then buttonList := buttonList( nextButton )
            else lag( nextButton ) := drop( nextButton )
        end

    let formClear = proc()                ! clear the form
        begin
            let drop := buttonList
            while drop ~= nil do
                begin
                    copy drop( savedArea ) onto limit screen at drop( xo ), drop( yo )
                    drop := drop( nextButton )
                end
            buttonList :=  nil
        end

    let formUpdate = proc( string s; pntr but )    ! change the message associated with a light button
        begin
            but( strip ) := s
            formShow( but )
        end
```

```
    let formMouse = proc( -> pntr )      ! return the light button clicked over
        begin
            let maxwell := locator()
            let found := false
            let actor := nil
            while ~found do
                begin
                    while ~maxwell( the.buttons ) (1) do maxwell := locator()
                    let check := buttonList
                    while check ~= nil do
                        begin
                            if check ( xo ) < maxwell( X.pos ) and
                                maxwell( X.pos ) < check (xo)+ check (xh) and
                                check ( yo ) < maxwell( Y.pos ) and
                                maxwell( Y.pos ) < check ( yo)+ check ( yh ) do
                                    { found := true; actor := check }
                            check := check ( nextButton )
                        end
                    maxwell := locator()
                end
            while  maxwell( the.buttons ) (1) do maxwell := locator()
            actor
        end

    let ffinished := false
    let fender = proc(); ffinished := true
    let formMonitor = proc()               ! monitor the form and do the action associated
        begin                              !     with the selected light button
            finished := false
            while ~ffinished do
                begin
                    let but = formMouse()
                    let buttonBox = limit screen to but (xh) by but (yh) at but (xo), but (yo)
                    not buttonBox onto buttonBox
                    but (action) ()
                    formShow( but )
                end
        end

    structure formPackage( proc( pntr) FormShow;
                proc( string,int,int,int,int,proc(),pntr -> pntr ) FormAdd;
                proc( pntr ) FormRemove;
                proc( string, pntr ) FormUpdate;
               proc( ) FormClear;
                proc( -> pntr ) FormMouse;
               proc( ) Fender;
               proc( ) FormMonitor)
    formPackage( formShow, formAdd, formRemove, formUpdate, formClear, formMouse,
                    fender, formMonitor )
  end ! form.generate
```

Appendix G. A Set of Interface Procedures to the Run-time Compiler.

```
! i) Provide sensible interface to the interactive compiler.
let compile = proc( cstring source ;cpntr record -> pntr )
    begin
        let f= "/tmp/COMPILE" ++ iformat( time() rem 10000 )
        let t= create(f,432)
        output t,source,"'n"
        close(f)
        compiler( t,record )
    end


! ii) Report Compiler Errors.
structure err.rec( cstring line.error, pntr next.error )
let reportErrors = proc(pntr C)
    begin
        let errs := C
        while errs ~= nil do { write errs( line.error ),"'n"; errs := errs( next.error ) }
        abort
    end


! iii) Make A Structure Definition.
let CltoStruc = proc( string theCl -> string)
    begin
        let outString := ""
        for i = 1 to length( theCl ) dooutString := outString ++
                ( if theCl ( i | 1 ) = "'n"   then ";"   else theCl ( i | 1 )  )
        outString := outString ( 1 | length( theCl ) -2 ) ++ ")"
        outString
    end


! iv) Return The Field Names.
let CltoNames = proc( string theCl -> *string)
    begin
        let tooStrings := vector 1::100 of ""
        let n := 0;          let j :=1;      let over := false
        while theCl ( j | 1 ) ~= "(" do j := j + 1
        while theCl ( j | 1 ) ~= ")" do
            begin
                let k :=j;       while theCl ( k | 1 ) ~= "'n" do k := k + 1
                j := k;          while theCl ( j | 1 ) ~= " " do j := j - 1
                n := n + 1
                tooStrings ( n ) := theCl ( j + 1 | k - j -1 )
                j := k + 1
            end
        let outStrings = vector 1::n of ""
        for i =1 to n do outStrings ( i ) := tooStrings ( i )
        outStrings
    end
```

```
! v) Return The Structure Name.
let CltoSname = proc( string theCl -> string)
    begin
        let j :=1
        while theCl ( j | 1 ) ~= "(" do j := j + 1
        theCl ( 1 | j - 1 )
    end


! vi) Return The Field Types.
structure typeList( string S; string R; pntr N )
let CltoTypes = proc( string theCl -> *pntr)
    begin
        let tooStrings := vector 1::100 of ""
        let n := 0
        let over := false
        let j :=1
        while theCl ( j | 1 ) ~= "(" do j := j + 1
        j := j + 1
        while theCl ( j | 1 ) ~= ")" do
            begin
                n := n + 1
                let k := j
                while theCl ( k | 1 ) ~= "'n" do k := k + 1
                 let l := k
                while theCl ( l | 1 ) ~= " " do l := l - 1
                tooStrings ( n ) := theCl ( j | l - j )
                 j := k + 1
            end
        let outPoints = vector 1::n of nil
        for i =1 to n do
            begin
                let ss :=  tooStrings ( i )
                let listMaker := proc( string inn -> pntr ); nullproc
                listMaker := proc( string inn -> pntr )
                    begin
                        let rest = inn( 2 | length(inn) - 1 )
                        case inn( 1 | 1 ) of
                            "*":  typeList( "*", rest, listMaker ( rest ) )
                             "c":  typeList ( "c", rest, listMaker ( rest ) )
                        default:  if inn( 1 | 3 ) = "pro"
                                      then typeList ( "proc", "", nil )
                                      else typeList ( inn, "", nil )
                    end
                outPoints ( i ) := listMaker ( tooStrings ( i ) )
            end
        outPoints
    end
```

Appendix H. CheckRecord - A procedure which checks if a structure has a particular string value in its fields.

```
let source := ""
let outInit = proc(); source := ""
let outLine = proc( string s ); source := source ++ s ++ "'n"

structure compileRecord ( proc( pntr, string -> bool ) compiledForm )

let ourTable := table()

let checkRecord = proc( pntr R; string IN -> bool)
  begin
    let CI = class.identifier( R )                                       ! A
    let ourProc := s.lookup( CI, ourTable )                              ! A
    if ourProc = nil do                                                  ! A
      begin
        let  theName   = CItoSname( CI )                                 ! C
        let theStructure= CItoStruc( CI )                                ! C
        let fieldNames  = CItoNames( CI )                                ! C
        let fieldTypes  = CItoTypes( CI )                                ! C

        let noFields := true                                             ! B
         for i = 1 to upb( fieldNames ) do                               ! B
              if fieldTypes (i)( S ) = "string" do noFields := false     ! B
        if noFields                                                      ! B
            then ourProc := compileRecord( proc( pntr RR; string SIN -> bool )   ! B
                         { write "No string fields in this record"; false } )    ! B
            else
              begin
                outInit()                                                ! C
                outLine ( "proc( pntr RR; string SIN -> bool )")         ! C
                outLine ( "     begin" )                                 ! C
                outLine ( "          structure " ++ theStructure )       ! C
                outLine ( "          let result := false" )              ! C

                for i = 1 to upb( fieldNames ) do                        ! C
                     if fieldTypes (i)( S ) = "string" do                ! C
                outLine ( "          if RR( " ++ fieldNames ( i ) ++ ") = SIN do result := true" )
                outLine ( "          result" )                           ! C
                outLine( "     end" )                                    ! C

                let emptyPackage =
                     compileRecord( pro c( pntr RR; string SIN -> bool ); nullproc )   ! D
                ourProc := compile( source, emptyPackage )               ! D
                if ourProc is err.rec do reportErrors( ourProc )         ! D
                s.enter( CI, ourTable, ourProc )                         ! A
            end
      end
    ourProc( compiledForm )( R, IN )                                     ! E
  end
```

Appendix I. An Object Copy Procedure.

```
let source := "";          let indent := ""
let outInit = proc(); source := ""
let outLine = proc( string s ); source := source ++ s ++ "'n"

structure compileRecord ( proc( pntr -> pntr ) compiledForm )
let objectCopy := proc( pntr theObject -> pntr ); nullproc
objectCopy := proc( pntr theObject -> pntr )
  begin
    outInit ()                                                  ! A
    outLine ( "proc( pntr A -> pntr )" )                        ! A - the main proc declaration

    let makeCode := proc( string PS; pntr P -> string ); nullproc
    makeCode := proc( string PS; pntr P -> string )             ! B - build a block
    if P = nil then "nil" else
      begin
        let CI = class.identifier( P )
        let  theName   = CItoSname( CI )
        let theStructure= CItoStruc( CI )
        let fieldNames  = CItoNames( CI )
        let fieldTypes  = CItoTypes( CI )

        let tSource := ""
        let tInit = proc(); tSource := ""
        let tStart = proc( string s ); { tSource := tSource ++indent ++  s }
        let tPart = proc( string s ); tSource := tSource ++ s
        let tEnd  = proc( string s ); tSource := tSource ++ s ++ "'n"
        let tLine = proc( string s ); { tSource := indent ++ tSource ++ indent ++ s ++ "'n" }
        tInit ()
        tLine( "" )
        indent := indent ++ "   "
        tLine ( "begin" )                                       ! B - block starts with begin
        indent := indent ++ "   "
        tLine ( "structure  " ++ theStructure )                 ! B - and structure definition

        for f = 1 to upb( fieldNames ) do
          begin
            let FnameF = PS ++ " (" ++ fieldNames( f ) ++ ") "
            tStart( "let X" ++ iformat(f) ++ " = " )            ! B - then start a line of the form
                                                                ! B - let X(1) = expression
            let parse := proc( pntr T ); nullproc
            parse := proc( pntr T )                             ! generate the expression
                 case T( S ) of
                   "int","bool","string","real","proc","pixel","#pixel","pic": tPart( FnameF )
                                                                ! E - scalars return field value
                  "c":   parse( T( N ) )                        ! E - constancy is ignored
```

```
"*":                                                   ! E -  vectors build a proc to generate
  begin                                                ! E - a block
    let vParse := proc( string VS; pntr VT -> string ); nullproc
    vParse:= proc( string VS; pntr VT -> string )
    if VT( S ) = "c" then vParse ( VS, VT( N ) ) else
      begin
        structure vRec( proc( pntr, pntr, string, proc( string, pntr ->
                    string ), proc( string, pntr -> string ) -> string ) vProg )
        indent := indent ++ "        "
        let vStart= "@ lwb( "++ VS ++" ) of "++ VT(R) ++"'n"++ indent ++" ["
        let vEnd = " ]"
        let vSource := ""
        let vLine = proc( string s ); Vsource := Vsource ++ s ++ "'n"
        vLine ( "proc( pntr VP, VTT; string VF; proc( string, pntr ->
                string ) MAKE.CODE, V.PARSE  -> string )")
        vLine ( "  begin")
        vLine ( "    structure " ++ theStructure )
        vLine ( "    let vOut := '"'"")
        vLine ( "    for i = lwb( VP( " ++ fieldNames( f ) ++
                "  ) ) to upb( VP( " ++ fieldNames( f ) ++ " ) ) do")

        let element = "VF ++ '"('" ++ iformat(i) ++ '")'""
        let makeUnit := proc( pntr MUT -> string ); nullproc
        makeUnit := proc( pntr MUT -> string )
            case MUT( S ) of
             "int","bool","string","real","proc","pixel","#pixel","pic":element ++ " ++ '", ""
             "c": makeUnit ( MUT( N ) )
             "pntr": "MAKE.CODE( " ++ element ++ ", VP(" ++
                       fieldNames( f ) ++ ")(i) ) ++ '",'n'""
             "*": "'"++ indent ++"  '" ++ VPARSE( "++ element ++
                  ", VTT ) ++ '",'n'""
        default: ""

        vLine ( "          vOut := vOut ++ " ++ makeUnit( VT( N ) ) )
        vLine ( "    vOut( 1 | length( vOut ) - 2 )")
        vLine ( "  end" )
        let vObj = compile( vSource,vRec( proc( pntr VP, VTT; string VF; proc(
              string, pntr -> string ) MAKECODE, VPARSE -> string ); nullproc ) )
        if vObj is err.rec do reportErrors( vObj )
        let vBody = vObj ( vProg )( P, VT( N ), VS, makeCode, vParse )
        indent := indent( 1 | length(indent) - 8 )
        vStart ++ vBody ++ vEnd
      end
  Part( vParse( FnameF, T ) )
end
```

```
              "pntr":                                  ! E - pointers build a proc to generate
               begin                                   ! E -  a block
                 structure pRec(proc(proc(string,pntr->string), pntr->string) pprog )
                 let pSource := ""
                 let pLine = proc( string s ); pSource := pSource ++ s ++ "'n"
                 pLine ("proc(proc(string,pntr->string)MAKE.CODE;pntr PP->string)")
                 pLine ( "  begin" )
                 pLine ( "    structure " ++ theStructure )
                 pLine ( "    MAKE.CODE( '"" ++ FnameF ++ "'",PP(" ++ fieldNames(f) ++ ") )"
                 pLine ( "  end" )

                 let dumObj = pRec( proc( proc( string, pntr -> string ) MAKECODE;
                                         pntr PP -> string ); nullproc )
                 let pObj = compile( pSource, dumObj )
                 if pObj is err.rec do reportErrors( pObj )
                 tEnd( "" )
                 tLine ( pObj ( pprog )( makeCode, P ) )
               end

          default: tEnd ( "" )                         ! E - unexpected!


          parse( fieldTypes(f) )                       ! B - call parse to make expression
          tEnd( "" )                                   ! B - end let X(1) = .. line
       end   ! of "for f" loop


     let final := theName ++ " ("
     for f = 1 to upb( fieldNames ) do final := final ++ "  x" ++ iformat(f) ++ ","
     let len= length( final )
     tLine ( final ( 1 | len -1 ) ++ ")" )             ! B - line of the form name(X1, .. )
     indent := indent( 1 | length( indent ) - 4 )
     tLine ( "end" )                                   ! B - end
     indent := indent( 1 | length( indent ) - 4 )
     tSource                                           ! B - return the whole block
   end   ! of makeCode


 outLine ( makeCode( "A", theObject ) )                ! A - the body of the main proc

 let c = compile( source, compileRecord( proc( pntr theObject -> pntr ); nullproc ) )
 if c is err.rec do reportErrors( c )                  ! A
 c( compiledForm )( theObject )                        ! A
end
```

Appendix J. The Object Copy Procedure in Action.

As a test object for the procedure, we take the object, *OBJECT*, created by the following:

> **structure** *Main*(**string** *Mstring*; **int** *Mint*; ***int** *Mvint*; **pntr** *Mpntr*)
> **structure** *Subsid*(**string** *Sstring*; ***bool** *Svbool*)
> **let** *OBJECT* := *Main*("Main", 1, @ 1 **of int** [1, 2, 3],
> *Subsid*("Subsid", @ 1 **of bool** [**true**, **false**]))

We will follow this through in steps which are numbered with an extra digit for each step which requires another level of procedure call.

1. *source* is initialised to the empty string.

2. The first line of the final procedure "`proc( pntr A -> pntr )`" is put into *source.*

3. *makeCode* is called with input parameter values, "A" and *OBJECT.*

3.1 *tSource* is initialised to be properly indented.

3.2 *tSource* is started with the lines:

```
begin
   structure MAIN(string Mstring; int Mint; *int Mvint; pntr Mpntr)
```

3.3 The variable, *FnameF*, is set to be "`A( Mstring )`".

3.4 A new line in *tSource* is started with the string "`let X1 =`".

3.5 *parse* is called with a pointer to the *typeList* associated with *Mstring* to put the expression for that field into *tSource*.

3.5.1 *parse* puts the value of *FnameF*, viz: "`A( Mstring )`", into *tSource* to achieve this.

3.6 *FnameF* is set to be "`A( Mint )`".

3.7 A new line in *tSource* is started with the string "`let X2 =`".

3.8 *parse* is called as in 3.5. It puts "`A( Mint )`" into *tSource.*

3.9 *FnameF* is set to be "`A( Mvint )`".

3.10 A new line in *tSource* is started with the string "`let X3 =`".

3.11 *parse* is called with a pointer to a *typeList* which shows that this field is a vector of integers.

3.11.1 *vParse* is called, with parameters, "`A( Mvint )`" and a pointer to the *typeList*, to put the appropriate vector expression in *tSource*.

3.11.1.1 *vParse* starts by setting *vEnd* to be "]" and *vStart* to be

```
"@ lwb( A( Mvint ) ) of int
          ["
```

3.11.1.2 *vSource* is initialised to the empty string and then has the following lines put in:

the procedure definition

```
begin
   structure Main( ... )
   let vout := ""
   for i = lwb(A(Mvint)) to upb(A(Mvint)) do
```

3.11.1.3 The variable, *element*, is set to the string

```
"VF ++ "("++iformat(i) ++")""
```

3.11.1.4 *vSource* then has a line started with "`vout:=vout++`". This is completed by a call to *makeUnit*, with the rest of the type information, i.e "int" as its argument.

3.11.1.4.1 *makeUnit* puts the value of *element*, above and "` ++","`" into *vSource.*

3.11.1.5 *vSource* is concluded with the lines

```
 vOut( 1 | length( vOut ) - 2 )
end
```

3.11.1.6 *vSource* is compiled and run, with input parameters: *OBJECT* - the input object; the "int" part of the type info; the string "`A(Mvint)`"; and the procedures *makeCode* and *vParse.*

3.11.1.6.1 The procedure in *vSource* is run and produces the string "`A(Mvint)(1),A(Mvint)(2),A(Mvint)(3)`" which is returned to *vBody.*

3.11.1.7 The contents of *vStart*, *vBody* and *vEnd* are returned as the result of *vParse.*

3.11.2 *parse* puts this string into *tSource.*

3.12 *FnameF* is set to be "`A( Mpntr )`".

3.13 A new line in *tSource* is started with the string "`let X4 =`".

3.14 *parse* is called with a pointer to the type info " pntr ".

3.14.1 A procedure is built in *pSource* which consists of the following lines:

```
proc(proc(string,pntr->string)MAKECODE; pntr PP ->string)"
   begin
     structure Main( .. )
     MAKECODE( "A(Mpntr)", PP(Mpntr) )
   end
```

3.14.2 The procedure is compiled and called with *makeCode* and *OBJECT* as its arguments.

3.14.2.1 The procedure makes a call to *makeCode* with "`A(Mpntr)`"and *OBJECT*(*Mpntr*) as its arguments.

3.14.2.1.1 *makeCode* begins a new *tSource* with the lines:

```
begin
  structure Subsid( ... )
```

3.14.2.1.2 *FnameF* is set to " `A(Mpntr)(Sstring)`".

3.14.2.1.3 A line in *tSource* is started "`let X1 =`".

3.14.2.1.4 *parse* is called to finish this line with the string
" `A(Mpntr)(Sstring)`"

3.14.2.1.5 *FnameF* is set to " `A(Mpntr)(Svbool)`"

3.14.2.1.6 A line in *tSource* is started "`let X2 =`"

3.14.2.1.7 *parse* is called to produce the vector expression.

3.14.2.1.7.1 *vParse* is called.

3.14.2.1.7.1.1 *vParse* builds the procedure:

```
proc( pntr VP, VTT,.....)
   begin
      structure Subsid( ...)
      let v.out := ""
      for i = lwb(VP(Svbool)) to upb(VP(Svbool)) do
      v.out := v.out++VP++"("++iformat(i)++")"++ ","
      v.out( 1 | length( vout) -2 )
   end
```

3.14.2.1.7.1.2 This procedure is compiled and run, producing as output:
" `A(Mpntr)(Svbool)(1), A(Mpntr)(Svbool)(2)`"

3.14.2.1.7.1.3 *vParse* returns as its result:

```
@ lwb(A(Mpntr)(Svbool)) of bool [ A(Mpntr)(Svbool)(1), A(Mpntr)(Svbool)(2) ]
```

3.14.2.1.7.2 *parse* puts this string into the second *tSource*

3.14.2.1.8 *makeCode* now builds the following string in *final*:

```
Subsid( X1, X2
```

3.14.2.1.9 This string, finished off with a "`)`" is put into *tSource*.

3.14.2.1.10 The second *tSource* is completed with and "`end`" line.
3.14.2.1.11 This call of *makeCode* ends by returning the second *tSource*.

3.14.3 This string which is now:

```
begin
   structure Subsid(....)
   let X1 =  A(Mpntr)(Sstring)
   let X2 = @ lwb(A(Mpntr)(Svbool)) of
              bool [ A(Mpntr)(Svbool)(1), A(Mpntr)(Svbool)(2) ]
   Subsid( X1, X2 )
end
```

is put into the first *tSource* as the end of the line which began (at 3.13) with "`let X4 =`".

3.15 The string variable, *final*, is set up to contain " `Main( X1, X2, X3, X4` ".

3.16 This together with a closing ")" are put into *tSource*.

3.17 *tSource* is completed with an "`end`" line.

3.18 *makeCode* quits returning the whole of *tSource*.

4. This string is then put into *source* which is now complete.

5. *source* is compiled. It now contains:

```
proc( pntr A -> pntr )
   begin
      structure MAIN(string Mstring; int Mint; *int Mvint; pntr Mpntr)
      let X1 = A( Mstring )
      let X2 = A( Mint )
      let X3 = @ lwb( A( Mvint ) ) of int
                  [ A(Mvint)(1),A(Mvint)(2),A(Mvint)(3) ]
      let X4 =
         begin
            structure Subsid(....)
            let X1 =  A(Mpntr)(Sstring)
            let X2 = @ lwb(A(Mpntr)(Svbool)) of
                  bool [ A(Mpntr)(Svbool)(1), A(Mpntr)(Svbool)(2) ]
            Subsid( X1, X2 )
         end
      Main( X1, X2, X3, X4 )
   end
```

6. This procedure is run and its output is returned as the output of *ObjectCopy*.

Bibliography

Copies of documents in this list may be obtained by writing to:

The Secretary,
Persistent Programming Research Group,
Department of Computing Science,
University of Glasgow,
Glasgow G12 8QQ
Scotland.

Books

Davie, A.J.T. & Morrison, R.
"Recursive Descent Compiling", Ellis-Horwood Press (1981).

Atkinson, M.P. (ed.)
"Databases", Pergammon Infotech State of the Art Report, Series 9, No.8, January 1982. (535 pages).

Cole, A.J. & Morrison, R.
"An introduction to programming with S-algol", Cambridge University Press, Cambridge, England, 1982.

Stocker, P.M., Atkinson, M.P. & Grey, P.M.D. (eds.)
"Databases - Role and Structure", Cambridge University Press, Cambridge, England, 1984.

Published Papers

Morrison, R.
"A method of implementing procedure entry and exit in block structured high level languages". Software, Practice and Experience 7, 5 (July 1977), 535-537.

Morrison, R. & Podolski, Z.
"The Graffiti graphics system", Proc. of the DECUS conference, Bath (April 1978), 5-10.

Atkinson, M.P.
"A note on the application of differential files to computer aided design", ACM SIGDA newsletter Summer 1978.

Atkinson, M.P.
"Programming Languages and Databases", Proceedings of the 4th International Conference on Very Large Data Bases, Berlin, (Ed. S.P. Yao), IEEE, Sept. 78, 408-419. (A revised version of this is available from the University of Edinburgh Department of Computer Science (EUCS) as CSR-26-78).

Atkinson, M.P.
"Progress in documentation: Database management systems in library automation and information retrieval", Journal of Documentation Vol.35, No.1, March 1979, 49-91. Available as EUCS departmental report CSR-43-79.

Gunn, H.I.E. & Morrison, R.
"On the implementation of constants", Information Processing Letters 9, 1 (July 1979), 1-4.

Atkinson, M.P.
"Data management for interactive graphics", Proceedings of the Infotech State of the Art Conference, October 1979. Available as EUCS departmental report CSR-51-80.

Atkinson, M.P. (ed.)
"Data design", Infotech State of the Art Report, Series 7, No.4, May 1980.

Morrison, R.
"Low cost computer graphics for micro computers", Software Practice and Experience, 12, 1981, 767-776.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"PS-algol: An Algol with a Persistent Heap", ACM SIGPLAN Notices Vol.17, No. 7, (July 1981) 24-31. Also as EUCS Departmental Report CSR-94-81.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Nepal - the New Edinburgh Persistent Algorithmic Language", in Database, Pergammon Infotech State of the Art Report, Series 9, No.8, 299-318 (January 1982) - also as EUCS Departmental Report CSR-90-81.

Morrison, R.
"S-algol: a simple algol", Computer Bulletin II/31 (March 1982).

Morrison, R.
"The string as a simple data type", Sigplan Notices, Vol.17,3, 46-52, 1982.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Progress with Persistent Programming", presented at CREST course UEA, September 1982, revised in "Databases - Role and Structure", see PPRR-8-84.

Morrison, R.
"Towards simpler programming languages: S-algol", IUCC Bulletin 4, 3 (October 1982), 130-133.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Problems with persistent programming languages", presented at the Workshop on programming languages and database systems, University of Pennsylvania. October 1982. Circulated (revised) in the Workshop proceedings 1983, see PPRR-2-83.

Atkinson, M.P.
"Data management", in Encyclopedia of Computer Science and Engineering 2nd Edition, Ralston & Meek (editors) January 1983. van Nostrand Reinhold.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Algorithms for a Persistent Heap", Software Practice and Experience, Vol.13, No.3, 259-272 (March 1983). Also as EUCS Departmental Report CSR-109-82.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"CMS - A chunk management system", Software Practice and Experience, Vol.13, No.3 (March 1983), 273-285. Also as EUCS Departmental Report CSR-110-82.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Current progress with persistent programming", presented at the DEC workshop on Programming Languages and Databases, Boston, April 1983.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"An approach to persistent programming", The Computer Journal, 1983, Vol.26, No.4, 360-365 - see PPRR-2-83.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"PS-algol a language for persistent programming", 10th Australian Computer Conference, Melbourne, Sept. 1983, 70-79 - see PPRR-2-83.

Morrison, R., Weatherill, M., Podolski, Z. & Bailey, P.J.
"High level language support for 3-dimension graphics", Eurographics Conference Zagreb, North Holand, 7-17, Sept. 1983. (ed. P.J.W. ten Hagen).

Cockshott, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J. & Morrison, R.
"POMS : a persistent object management system", Software Practice and Exerience, Vol.14, No.1, 49-71, January 1984.

Kulkarni, K.G. & Atkinson, M.P.
"Experimenting with the Functional Data Model", in Databases - Role and Structure, Cambridge University Press, Cambridge, England, 1984.

Atkinson, M.P. & Morrison, R.
"Persistent First Class Procedures are Enough", Foundations of Software Technology and Theoretical Computer Science (ed. M. Joseph & R. Shyamasundar) Lecture Notes in Computer Science 181, Springer Verlag, Berlin (1984).

Atkinson, M.P., Bocca, J.B., Elsey, T.J., Fiddian, N.J., Flower, M., Gray, P.M.D. Gray, W.A., Hepp, P.E., Johnson, R.G., Milne, W., Norrie, M.C., Omololu, A.O., Oxborrow, E.A., Shave, M.J.R., Smith, A.M., Stocker, P.M. & Walker, J.
"The Proteus distributed database system", proceedings of the third British National Conference on Databases, (ed. J. Longstaff), BCS Workshop Series, Cambridge University Press, Cambridge, England, (July 1984).

Atkinson, M.P. & Morrison, R.
"Procedures as persistent data objects", ACM TOPLAS 7, 4, 539-559, (Oct. 1985) - see PPRR-9-84.

Morrison, R.,Bailey, P.J., Dearle, A., Brown, P. & Atkinson, M.P.
"The persistent store as an enabling technology for integrated support environments", 8th International Conference on Software Engineering, Imperial College, London (August 1985), 166-172 - see PPRR-15-85.

Atkinson, M.P. & Morrison, R.
"Types, bindings and parameters in a persistent environment", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 1-24 - see PPRR-16-85.

Davie, A.J.T.
"Conditional declarations and pattern matching", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 278-283 - see PPRR-16-85.

Krablin, G.L.
"Building flexible multilevel transactions in a distributed persistent environment, proceedings of Data Types and Persistence Workshop, Appin, August 1985, 86-117 - see PPRR-16-85.

Buneman, O.P.
"Data types for data base programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 291-303 - see PPRR-16-85.

Cockshott, W.P.
"Addressing mechanisms and persistent programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 363-383 - see PPRR-16-85.

Norrie, M.C.
"PS-algol: A user perspective", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 399-410 - see PPRR-16-85.

Owoso, G.O.
"On the need for a Flexible Type System in Persistent Programming Languages", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 423-438 - see PPRR-16-85.

Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. & Dearle, A.
"A persistent graphics facility for the ICL PERQ", Software Practice and Experience, Vol.14, No.3, (1986) - see PPRR-10-84.

Atkinson, M.P. and Morrison R.
"Integrated Persistent Programming Systems", proceedings of the 19th Annual Hawaii International Conference on System Sciences, January 7-10, 1986 (ed. B. D. Shriver), vol IIA, Software, 842-854, Western Periodicals Co., 1300 Rayman St., North Hollywood, Calif. 91605, USA - see PPRR-19-85.

Atkinson, M.P., Morrison, R. and Pratten, G.D.
"A Persistent Information Space Architecture", proceedings of the 9th Australian Computing Science Conference, January, 1986 - see PPRR-21-85.

Kulkarni, K.G. & Atkinson, M.P.
"EFDM : Extended Functional Data Model", The Computer Journal, Vol.29, No.1, (1986) 38-45.

Buneman, O.P. & Atkinson, M.P.
"Inheritance and Persistence in Database Programming Languages"; proceedings ACM SIGMOD Conference 1986, Washington, USA May 1986 - see PPRR-22-86.

Morrison R., Dearle, A., Brown, A. & Atkinson M.P.; "An integrated graphics programming environment", Computer Graphics Forum, Vol. 5, No. 2, June 1986, 147-157 - see PPRR-14-86.

Atkinson, M.G., Morrison, R. & Pratten G.D.
"Designing a Persistent Information Space Architecture", proceedings of Information Processing 1986, Dublin, September 1986, (ed. H.J. Kugler), 115-119, North Holland Press.

Brown, A.L. & Dearle, A.
"Implementation Issuses in Persistent Graphics"; The Association for Computing Machinery, 11 West 42nd St., New York, NY 10036; University Computing, Vol. 8, NO. 2, (Summer 1986) - see PPRR-23-86.

Internal Reports

Morrison, R.
"S-Algol language reference manual", University of St Andrews CS-79-1, 1979.

Bailey, P.J., Maritz, P. & Morrison, R.
"The S-algol abstract machine", University of St Andrews CS-80-2, 1980.

Atkinson, M.P., Hepp, P.E., Ivanov, H., McDuff, A., Proctor, R. & Wilson, A.G.
"EDQUSE reference manual", Department of Computer Science, University of Edinburgh, September 1981.

Hepp, P.E. and Norrie, M.C.
"RAQUEL: User Manual", Department of Computer Science Report CSR-188-85, University of Edinburgh.

Norrie, M.C.
"The Edinburgh Node of the Proteus Distributed Database System", Department of Computer Science Report CSR-191-85, University of Edinburgh.

In Preparation

Atkinson, M.P. & Buneman, O.P.
"Database programming languages design", submitted to ACM Computing Surveys - see PPRR-17-85.

Theses

The following Ph.D. theses have been produced by members of the group and are available from the address already given,

W.P. Cockshott
Orthogonal Persistence, University of Edinburgh, February 1983.

K.G. Kulkarni
Evaluation of Functional Data Models for Database Design and Use, University of Edinburgh, 1983.

P.E. Hepp
A DBS Architecture Supporting Coexisting Query Languages and Data Models, University of Edinburgh, 1983.

G.D.M. Ross
Virtual Files: A Framework for Experimental Design, University of Edinburgh, 1983.

G.O. Owoso
Data Description and Manipulation in Persistent Programming Languages, University of Edinburgh, 1984.

Persistent Programming Research Reports

This series was started in May 1983. The following list gives those produced and those planned plus their status at 10th April 1987. Copies of documents in this list may be obtained by writing to the address already given.

PPRR-1-83	The Persistent Object Management System - Atkinson, M.P., Chisholm, K.J. and Cockshott, W.P.	£1.00
PPRR-2-83	PS-algol Papers: a collection of related papers on PS-algol - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-4-83	The PS-algol reference manual - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R	Presently no longer available
PPRR-5-83	Experimenting with the Functional Data Model - Atkinson, M.P. and Kulkarni, K.G.	£1.00
PPRR-6-83	A DBS Architecture supporting coexisting user interfaces: Description and Examples - Hepp, P.E.	£1.00
PPRR-7-83	EFDM - User Manual - K.G.Kulkarni	£1.00
PPRR-8-84	Progress with Persistent Programming - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-9-84	Procedures as Persistent Data Objects - Atkinson, M.P.,Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£1.00
PPRR-10-84	A Persistent Graphics Facility for the ICL PERQ - Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. and Dearle, A.	£1.00
PPRR-11-85	PS-algol Abstract Machine Manual	£1.00
PPRR-12-86	PS-algol Reference Manual - third edition	£2.00
PPRR-13-85	CPOMS - A Revised Version of The Persistent Object Management System in C - Brown, A.L. and Cockshott, W.P.	£2.00
PPRR-14-86	An Integrated Graphics Programming Environment - second edition - Morrison, R., Brown, A.L., Dearle, A. and Atkinson, M.P.	£1.00
PPRR-15-85	The Persistent Store as an Enabling Technology for an Integrated Project Support Environment - Morrison, R., Dearle, A, Bailey, P.J., Brown, A.L. and Atkinson, M.P.	£1.00
PPRR-16-85	Proceedings of the Persistence and Data Types Workshop, Appin, August 1985 - ed. Atkinson, M.P., Buneman, O.P. and Morrison, R.	£15.00
PPRR-17-85	Database Programming Language Design - Atkinson, M.P. and Buneman, O.P.	£3.00
PPRR-18-85	The Persistent Store Machine - Cockshott, W.P.	£2.00
PPRR-19-85	Integrated Persistent Programming Systems - Atkinson, M.P. and Morrison, R.	£1.00
PPRR-20-85	Building a Microcomputer with Associative Virtual Memory - Cockshott, W.P.	£1.00
PPRR-21-85	A Persistent Information Space Architecture - Atkinson, M.P., Morrison, R. and Pratten, G.D.	£1.00
PPRR-22-86	Inheritance and Persistence in Database Programming Languages - Buneman, O.P. and Atkinson, M.P.	£1.00
PPRR-23-86	Implementation Issues in Persistent Graphics - Brown, A.L. and Dearle, A.	£1.00
PPRR-24-87	Using a Persistent Environment to Maintain a Bibliographic Database - Cooper, R.L., Atkinson, M.P. & Blott, S.M.	£1.00
PPRR-25-87	Applications Programming in PS-algol - Cooper, R.L.	£1.00
PPRR-26-86	Exception Handling in a Persistent Programming Language - Philbrow, P & Atkinson M.P.	£1.00
PPRR-27-86	A Context Sensitive Addressing Model - Hurst, A.J.	£1.00
PPRR-28-86b	A Domain Theoretic Approach to Higher-Order Relations - Buneman, O.P. & Ochari, A.	£1.00

PPRR-29-86	A Persistent Store Garbage Collector with Statistical Facilities - Campin, J. & Atkinson, M.P.	£1.00
PPRR-30-86	Data Types for Data Base Programming - Buneman, O.P.	£1.00
PPRR-31-86	An Introduction to PS-algol Programming - Carrick, R., Cole, A.J. and Morrison, R.	£1.00
PPRR-34-87	Binding Issues in Database Programming - Atkinson, M.P., Morrison, R., Cooper, R.L. and Abderrahmane, D.	£1.00