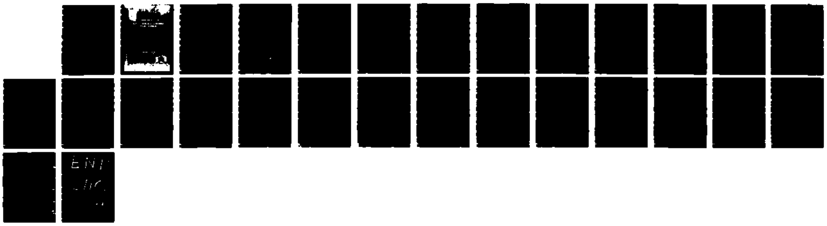
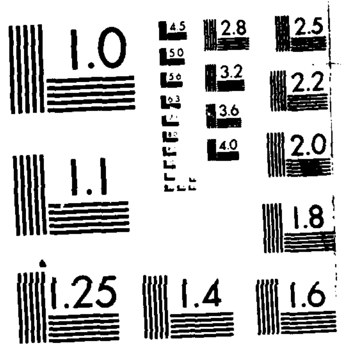
AD-A168 105 FLEX PASCAL: AN IMPLEMENTATION OF THE ISO-PASCAL PROGRAMMING LANGUAGE(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) K CURTIS ET AL. DEC 85 1/1

UNCLASSIFIED RSRE-MEMO-3908 DRIC-BR-99099 F/G 9/2 NL




1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

RSRE
MEMORANDUM No. 3908

ROYAL SIGNALS & RADAR ESTABLISHMENT

FLEX PASCAL: AN IMPLEMENTATION OF THE ISO-PASCAL PROGRAMMING LANGUAGE

Authors: K Curtis, P D Hammond
and P D Taylor

MEMORANDUM No. 3908

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 3908

Title : FLEX Pascal : An implementation of the ISO-Pascal programming language

Authors : K. Curtis, P.D. Hammond & P.D. Taylor

Date : December 1985

Summary

The Flex-Pascal compiler runs on the RSRE Flex computer architecture, which is currently available on the ICL Perq 2 workstation. Flex-Pascal conforms to the ISO 7185 (BS 6192) standard for Pascal and provides (as one extension) a separate-compilation facility. This paper discusses the use of ISO 7185 and the BSI Pascal Validation Suite, and criticises areas of Pascal which, by their nature, have proved particularly difficult to implement.

Contents

1. Introduction

2. The Structure of the Compiler

3. The ISO Standard and BSI Validation Suite

4. Difficulties of Implementation
   - 4.1 Sets
   - 4.2 Records
   - 4.3 For Loops
   - 4.4 Formal Parameter Specification

5. Separate Compilation

6. External Files

7. Comments on Possible Extensions to ANSI-Pascal

8. Summary

Acknowledgement

References

Appendix
   The Implementation of Sets

1 Introduction

The Flex-Pascal compiler runs on the RSRE Flex computer architecture [Currie:81, 83] [Foster:82], which is currently available on the ICL Perq 2 workstation. Flex-Pascal conforms to the ISO 7185 standard [BSI:82] for Pascal and provides (as one extension) a separate-compilation facility. This paper discusses the use of ISO 7185 and the BSI Pascal Validation Suite, and criticises areas of Pascal which, by their nature, have proved particularly difficult to implement.

Section 2 describes the overall structure of the Flex-Pascal compiler. Section 3 discusses the ISO standard for Pascal and the BSI Pascal Validation Suite which was used to provide a test harness for the compiler.

Section 4 describes the implementation of particular areas of ISO-Pascal and difficulties caused by the nature of the standard. Suggestions are made as to how matters could be improved.

As an extension to ISO-Pascal, Flex-Pascal provides a separate compilation facility; this is discussed briefly in section 5. Section 6 describes external files, and the way that they are treated on Flex.

A review of the extension proposals for ANSI-Pascal and a discussion of their impact for the Flex compiler is contained within section 7. Section 8 summarizes the benefits of the ISO standard.

2 The Structure of the Compiler

The Flex-Pascal compiler is written in Algol68 and is composed of about fifty modules i.e. separately-compiled units. As is usual on Flex, the Pascal compiler will indicate compilation errors by invoking the editor and arranging to display messages at appropriate places in the text. Thus, using the editor, one can step from one mistake to the next making corrections there and then.

If a Pascal program fails at run-time then the Flex system can be used to diagnose the failure. A trace of procedure calls is given; for each level the approximate place of the failure in the text is marked and local values are available. These are actual values and can be manipulated, e.g. variables can be examined, arrays indexed and procedures called.

The Flex-Pascal compilation system is two-pass with the intermediate code consisting of a Ten15 node tree. The two passes are illustrated by the diagram below

$$\begin{tikzcd} \boxed{\text{Pascal text}} \arrow[r, "\text{Pascal compiler}", "\text{(front end)}"'] & \boxed{\text{Ten15 tree}} \arrow[r, "\text{Ten15 translator}"] & \boxed{\text{code}} \end{tikzcd}$$

Ten15 is a strongly-typed, high-level program representation suitable for compiling many different languages into. The Flex-Ada compiler also produces Ten15 and it is intended that all new Flex compilers will do likewise. Thus, Ten15 facilitates mixed-language programming. It is also possible, although less easy, to call Algol68 procedures from Ten15 and vice-versa.

One of Ten15's strong points is the lack of any implicit coercions: every dereference and range change must be explicit. Consequently stringent checks are made by the Ten15 translator on the output from the front end of the Pascal compiler. This forces the correct implementation of difficult areas in Pascal by preventing any 'cheating'.

The compiler is syntax driven, being accessed through procedure calls embedded in the syntax. The syntax given in the ISO standard had to be transformed into LL(1), and use was made of SID [Foster:68], [Goodenough:85]. An LL(1) grammar can be used to analyse input by just looking one symbol ahead with no back-tracking. The output from SID consists of a syntax analyser which manipulates a set of stacks and calls the embedded procedures. It is these procedures that constitute the body of the compiler.

3 The ISO Standard and BSI Validation Suite

The Standard

The implementation of Flex-Pascal was required to conform to ISO standard for Pascal. The document used was the British Standards Institution (BSI) specification for the Computer Programming Language Pascal (BS 6192) which has been adopted by the International Organization for Standardization (ISO) to form the technical content of the standard ISO 7185. The BSI provides a Pascal quality control package as part of its compiler validation service, and an important element in this package is the Pascal Validation Suite (PVS). Version 4.0 of the PVS was used to form a test harness for the Flex-Pascal compiler.

The standard and the validation suite both refer to processors not compilers. A processor is a system that accepts a program or input, prepares it for execution and executes it with data to produce results. For example, the Flex-Pascal compiler in itself is only part of a processor of which the run-time system, the computer (ICL Perq) and operating system (Flex) form the rest. The standard applies to both processors and programs, so that any conforming program should compile and run on any conforming processor. This means that Pascal programs should be highly portable between conforming processors.

The standard defines two levels of compliance, level 0 and level 1, the difference being that conformant array parameters are only present in level 1. Conformant arrays allow a procedure to have variable size array parameters.

The standard defines errors as "A violation by a program of the requirements of this standard that a processor is permitted to leave undetected". That is, a processor need not detect all errors. However, the documentation that accompanies each processor is required to state which errors are always detected. There are fifty-nine errors specified in the standard.

Whilst the standard goes a long way to defining precisely the syntax and semantics of Pascal, there are areas where it is not clear what the interpretation should be. For example when defining the meaning of the statement

```
read(f, v1, ..., vn)
```

where f denotes a file, the standard says that this is equivalent to

```
begin read(f, v1); ...; read(f, vn) end
```

Now suppose that the statement is

```
read(af[i], i, n)
```

where af is an array of files. Either the reading of i could change the file n is read from, or the file could be accessed once and i and n read from the same file.

Similar problems occur in the definitions of "write", and the transfer procedures "pack" and "unpack". In these areas where the meaning in the standard is ambiguous, the Flex-Pascal compiler is nearer to the ANSI standard (ANSI/IEEE 770X3.97-1983) which has gone some way towards defining these areas more precisely.

The Validation Suite

This is a suite of Pascal programs designed to test whether or not a processor conforms to the ISO standard for the language Pascal. It is split up into eight main sections:

1. Conformance
This section contains programs which all conform to the standard. Each program tests one or more features of the language. For a processor to be validated it has to compile and run successfully all the programs in this section.

2. Deviance
Each program in this section does not conform to the standard in some way. Therefore a processor should fail to compile and run every program; failures normally occur at compile time. The sorts of things tested for are extensions to the standard, failure to check some feature, or some fault common amongst unvalidated processors.

3. Implementation Defined
These are areas in the standard where it is not defined what every processor should do, but the behaviour of each individual processor must be documented. The programs in this section each test such an area and should compile and run successfully. Each program prints out details of what the processor does. Examples of such features are the ordinal number corresponding to each value of type char, and default Total Widths when writing to Pascal Textfiles.

4. Implementation Dependent
Some areas in the standard need not even be specified for a particular processor. A program which depends on such a feature does not conform to the standard. The programs in this section however, give details about what happens in particular instances. Mostly these are the order of evaluation of items within a single statement, for example the order of evaluation of array indices.

5. Error Handling

The programs in this section test which errors, as defined in the standard, are detected and which are not. Each test consists of two programs: a Pretest which must compile and run successfully, and a program which is similar to the Pretest except that it contains an occurrence of the error being tested for. A processor must fail to run all programs that contain the errors which are claimed to be detected.

6. Quality

The tests in this section examine the "quality" of a processor. Whether a processor is successful in compiling and running this type of program makes no difference to the validation. The tests examine such things as the accuracy of real numbers, restrictive limits, for example on the depth of nesting procedures, or performance tests that can be timed and compared to other processors.

7. Extensions

These tests contain extensions to the standard that have been approved by the Pascal Users' Group. However programs containing these extensions do not conform to the standard and a processor should reject these programs. A validated processor may have a switch to allow/disallow extensions. The extensions in Flex-Pascal, which include a separate compilation system, can thus be enabled at compile time.

8. Level 1

This section contains programs which use conformant arrays. A processor that is compliant at level 0 should reject all of these programs whereas a processor compliant at level 1 should treat each test according to its subclass (which is one of the previous sections).

Whilst the suite tests many aspects, it obviously cannot test everything, yet it is tempting to use it as the only or major method of testing. This is a mistake as a processor may pass all the tests in the suite and still contain many errors, as was found with the Flex-Pascal compiler.

4.0 Difficulties of Implementation

There were four specific areas of the standard which caused some difficulty in the implemention of the Flex-Pascal compiler and which could, in our opinion, be improved upon. These areas are: the "width" and implementation of sets; checking the accesses to the variant parts of a record variable; the declaration and scope of the loop counter in for-loops; and the specification of procedural parameters.

4.1 Sets

The values of a Pascal set-type are defined as the powerset of another (ordinal) type, known as its base-type. For example, if **S** is the set-type whose base-type is the subrange 1..3, then the eight possible values of **S** are:

{1,2,3}, {1,2}, {1,3}, {2,3}, {1}, {2}, {3} and {}.

In general, if a base-type has n values, then the set-type has 2^n values. The operations applicable to all set-types are: union, intersection, difference, equality, inequality, membership, and set inclusion (+, *, -, =, <>, **in** , <=, >=). Set values can be constructed by enclosing in square brackets a list of single values or ranges to indicate those members present. Set-types may be designated **packed** to indicate that storage is to be optimized at the expense of execution speed of set-operations.

Originally Wirth [Wirth:71] implemented Pascal sets as one machine word, with each bit indicating whether an element was present in a set or not. Consequently, the cardinality of a set was limited by the length of a machine word (60-bits) and the least possible ordinal-value of a member was 0. Such restrictions made operations on these "narrow" sets very efficient.

The ISO standard relaxes these constraints and "widens" sets such that it is possible to declare sets whose elements may lie anywhere in the integer range (e.g. a set may contain elements ranging from -10 to -1, or even **-maxint** to **maxint**). If an implementation were to be based on a simple extension of Wirth's method and represent sets as n-bits (for a base-type of n values), then this would clearly require a ridiculous (if not impossible) amount of store for large n. Consequently, a new approach must be adopted if sets are to be implemented according to the standard. Such implementations will need to optimise the amount of store required for each particular set-value. This is likely to lead to inefficiences (in time and/or space) over earlier methods: let us consider two possibilities and compare them with Wirth's implementation of "narrow" sets.

A value of a set-type may be implemented as a (possibly ordered) list; it is then a simple matter to cater for "wide" sets. Unfortunately the implementation of set-operations becomes more complex. Union, for example, does not just involve concatenating two lists, but also ensuring that no duplication of elements occurs. This extra complexity will undoubtably lead to a reduction in the execution speed of set-operations. However, this is not the greatest drawback of this method; to represent a set with *m* members would require typically 2 words per member (1 word for the ordinal-value of the member, and 1 word for the address of the next member), a total of 2**m* words. Using Wirth's method, just 1 word is required to store a set, irrespective of the number of members in it. Using a list to represent "wide" sets is therefore slower than using Wirth's "narrow" sets, and certainly much more expensive in terms of storage.

A second method of implementing "wide" sets, and the one that has been adopted for Flex-Pascal, is to use one-dimensional arrays of booleans. These boolean arrays observe certain conventions:

(a) The empty set ({}) has bounds 1:0;

(b) A non-empty set has upper bound equal to the ordinal-value of the greatest member;

(c) A non-empty set has lower bound equal to the ordinal-value of the least member;

Note that the upper and lower bounds are *not* defined as the greatest and least ordinal-values of the base-type. Consequently, two set values of the same base-type will not (in general) have the same bounds.

The indication of the least and greatest (actual) set members is crucial to the implementation method. All set-operations must preserve these properties. By making the size of an array cover only the range of members that are actually in the set, and provided this actual range is not too ambitious, it is possible to represent a set whose base-type may cover any range of values.

This method places no restrictions on the cardinality of sets nor on the least ordinal-value of set members, but it does involve a practical limit to the range of ordinal-values of (actual) set members. It is found that for ranges of up to 1024, Flex-Pascal sets work adequately. Greater limits are possible, but a Pascal program may require a large amount of store in order to run. In terms of storage, this method is comparable with Wirth's, although set-operations are again more complex (see below), and therefore, much more time consuming. However, it is hoped that in the near future, logical operations will be available in Ten15 to perform **AND**, **OR** and **NOT** on boolean arrays, and thus lead to an improvement in speed.

As this second method is the one chosen for Flex-Pascal, it will be illustrated further. Consider the type set-of-integer; there are $2^{(2*\mathbf{maxint}+1)}$ different values of this set-type, of which one example is:

{-10, -6, -3, 0, 1, 4}

This is represented as the boolean array with bounds -10:4 whose value is:

element	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	+1	+2	+3	+4
value	T	F	F	F	T	F	F	T	F	F	T	T	F	F	T

To determine whether a particular value, *v*, with ordinal-value *n*, is a member of a set represented by an array **A**, one first examines the bounds of **A**; if *n* lies outside those bounds then *v* is not a member, otherwise the boolean value $\mathbf{A}_n$ yields the answer.

The union of two sets (S = S1 ∪ S1, where S, S1 and S2 are represented by arrays A, A1 & A1) is performed as follows: if S1 = {} then A = A2, if S2 = {} then A = A1, otherwise a new array, A3, is created whose bounds are chosen to cover the combined ranges of A1 and A2 (i.e. **LWB** A3 = minimum(**LWB** A1, **LWB** A2), **UPB** A3 = maximum(**UPB** A1, **UPB** A2)). All the elements of A3 are initialized to **FALSE**. The region of A3 that corresponds to A1 is then made equal to A1, and the region in A3 that corresponds to A2 is **OR**-ed with A2. The results of all set-operations must maintain the properties (a), (b) and (c) given above, and in some cases the final array (A3) may have to be trimmed to achieve this. In the case of set union however, this is never necessary, as the lower and upper elements of A3 must both be members of S, and so A = A3.

Other set-operations follow in an obvious manner. Appendix A contains the text of an Algol68-RS [Woodward:83] module that defines the data structure used for sets and describes the algorithms used for set-operations.

To summarise, one finds that the Pascal programmer nearly always wants the efficiencies of "narrow" sets, and is content to live with - and often does not notice - their restrictions. In fact, this preference is so strong amongst the Pascal fraternity, that there is an open conspiracy to ignore the ISO definition of "wide" sets. This conspiracy even extends to the BSI's own Pascal Validation Suite, where programs that test whether negative elements are allowed in sets are classified not as conformance, but as *quality* tests. This pretence is understandable, both from the implementors' and users' point of view. Considering the strength of feeling for "narrow" sets, it is difficult to imagine why the BSI's Technical Committee ever decided to opt for anything else.

The authors of this paper suggest that the ISO standard be modified to define clearly an implementation-defined maximum cardinality for sets, and that the least ordinal-value of a set member be 0. Following this amendment, "narrow" sets would get the official status that they so clearly deserve.

4.2 Records

A Pascal record comprises a fixed part and a variant part each of which can be empty. The fixed part consists of a number of components of different types and the variant part is a group of objects, known as variants, each of which is like a Pascal record. The behaviour of the variant part is such that at any time one, and only one, of the variants is active. It is only when a variant is active that its components are available for use as fields of the record.

Furthermore, variant parts can take one of two different formats which affect the way components are accessed. The first sort, a strong variant part, is one with a tag identifier. Strong variants are made active by assigning an appropriate value to the tag identifier. The second sort, a weak variant part, has no tag identifier and variants are made active by the *act* of accessing one of its components.

The rules on accessing components of weak variant parts mean that it is never an error to access any component. However, this does not mean that errors cannot occur whilst accessing such components, since an undefined value is automatically used if the access which activates a variant occurs on the right-hand side of an assignment statement. Strong variant parts have much more stringent rules for accessing components.

In the Flex-Pascal compiler each variant is a reference to a structure of the components of that variant. A variant part then consists of the tag and a union of void plus all the variants. Unions are dynamic modes specified by a list of possible modes that could be present. Initially the union has mode void, denoting that none of the variants is active. When a variant is active the union has the mode of that variant. For strong variant parts each access to a component involves a check on the union, asserting that the required variant is active. When the tag identifier is assigned to, space is created to hold the components of the new variant, and a reference to the space is assigned to the union. This ensures that the components of the new variant are initialised properly. For weak variant parts no checks can be made but new variants are made active by the same process.

One requirement of the standard which is difficult to implement is the following: "it is an error unless a variant is active fcr the entirety of each reference and access to each component of the variant". This means that it is an error to make a different variant active when a reference to a component of the previously active variant exists. The following example illustrates the problem :-

```
program variantproblem(output);

type range = 1..2;
     rec = record a:integer;
                  case tag:range of
                  1:(b:integer);
                  2:(c:real)
           end ;

var r:rec;

procedure p (var y:integer;var z:rec);
begin
  z.tag := 2;
  writeln(y)
end;

begin
  r.tag := 1;
  r.b := 24;
  p(r.b,r)
end.
```

When the procedure "p" is called a reference to "b" - a component of the active variant - is established and passed as a parameter. The whole record variable is also passed by reference and the first statement of "p" changes which variant is active by assigning to the tag identifier. This is an error because via "y" a reference exists to a component of the previously active variant. The error is difficult to detect since it depends on the actual parameters of this call of "p".

To detect this error it is necessary to know whether a reference is a component of a variant as well as which record variable it is associated with. The ISO standard forbids an actual **var** parameter from being a tag identifier which prevents a more difficult version of this problem from arising.

The Flex-Pascal compiler does not detect this error. But because the space used by the new variant is disjoint from the space used by the old one, "y" retains its value and 24 will be output. The established reference keeps the space used by the old variant alive for the entirety of the reference.

4.3 For Loops

The following is the syntax of ISO-Pascal **for** loops:

```
for-statement ="for" identifier ":=" for-list "do" statement.

for-list = expression "to" expression |
           expression "downto" expression.
```

The identifier that serves as the loop counter is a variable and is declared - as are all Pascal objects - in the declaration part that precedes the statement part of a block. Consequently the identifier is of a wider scope than that of the loop, and this causes a few problems as will be seen.

The loop counter must be of an ordinal-type, and the expressions in the for-list must be compatible with it. The occurrence of the reserved word **to** (**downto**) in the for-list means that the loop counter takes a sequence values starting at the first expression and increasing (decreasing) by one until the second expression is reached; for each value in the sequence the repeated statement is executed once. However, if the first expression should have a value greater (less) than that of the second, then the repeated statement is never executed.

It is the intention of the ISO standard that the loop counter shall not vary within the repeated statement. This is not easy to ensure at run time, so a few "behavioural" rules are introduced to enable invariance to be checked at compile time. In order for a variable to be used as loop counter, it must satisfy the "behaviour" rules, which may be summarised as follows:

1. The variable must be declared locally;

2. There must be no "threat" to the variable in the repeated statement nor in any procedure or function.

A "threat" to a variable is anything that *could* change its value, for example: being assigned to, or being an actual **var** parameter or being used as a loop counter.

To check that a variable obeys these rules is a tedious matter for a compiler. Because a loop variable is declared in the same way as any other variable, and for that matter, outside the loop it is permitted to assume the role of an "ordinary" variable, all variables are potential loop counters. Thus, in the case of the front end of the Flex-Pascal compiler, the behaviour of *all* variables must be monitored in case they are used as loop counters. This adds an unnecessary compile-time overhead.

Languages such as Algol68 [Algol68:76], and more recently, Ada [Ada:83], restrict the scope of the loop counter by making the for-statement serve as the declaration of a *constant* whose scope *is* the for-statement. This solves all the problems but perhaps would not suit the Pascal doctrine of not mixing declarations and statements. But something more befitting Pascal would be to have a section in the declarations part specifically for loop counters; the Pascal programmer would then state his intentions for a variable by declaring it in the appropriate section. Behavioural checks then need only be performed on variables declared in the special loop-counter section.

Another curiosity of ISO-Pascal is that the value of the variable used as the loop counter shall be undefined after the execution of a for-statement (except if left by a **goto**). This means that, following the normal completion of the for-statement, the loop counter shall not propagate any information. This is tantamount to admitting that its scope should be restricted to the for-statement. The loop counter may propagate information when jumping out of the loop, but this does not detract from our argument; this propagation is merely an unpleasant concession to facilitate implementation.

4.4 Formal Parameter Specification

When Wirth first described Pascal, its syntax allowed procedural parameters. However, the programmer was unable to specify them with sufficient precision in order to prevent insecurities and misuse. For example, the following program was syntactically correct:

```
program example1;

   procedure p1 (i : integer); begin end;

   procedure p2 (var j : integer); begin end;

   procedure p3; begin end;

   procedure p4 (var i,j : integer); begin end;

   procedure p5 (procedure q1); begin q1 end;

begin p5(p1); p5(p2); p5(p3); p5(p4) end.
```

One of the aims of the BSI in creating its standard Pascal was to rid it of all the ambiguities and insecurities that plagued its precursors.

To this end a new syntax for formal parameters was introduced, an abstract of which is the following:

```
procedure-declaration > procedure-heading ";" procedure-block.

procedure-heading = "procedure" identifier [formal-parameter-list].

formal-parameter-list = "(" formal-parameter-section
                            { ";" formal-parameter-section }
                        ")".

formal-parameter-section > identifier-list ":" type-identifier |
                           "var" identifier-list ":" type-identifier |
                           procedure-heading.
```

The program "example1" does not conform syntactically to ISO-Pascal, and so would be rejected by a compiler. Moreover, as all formal procedural parameters are now specified to the same extent as procedures themselves, the compiler checks the congruency of formal and actual procedural parameters. The objective has been achieved: no more insecurities. Unfortunately, the same syntax (procedure-heading) has been used in procedure-declaration as well as formal-parameter-section. Consequently, when specifying a procedural parameter, q, the programmer must supply superfluous names for the formal parameters of q, if it has any. The following example will illustrate this.

```
program example2;

  procedure p6 (i : integer); begin end;

  procedure p7 (procedure q2 (j : integer)); begin q2(42) end;

begin p7(p6) end.
```

In "example2" the formal parameter "q2", is specified by naming its parameter as "j" of type "integer". However when calling "p7", the actual parameter "p6" does not need to have matching parameter names with "q2". Yet what must match is the number of parameters of "p6" and "q2", and their corresponding types. Futhermore, the sections in their formal parameter lists must also match. The example below breaks this last rule, and so would fail to compile.

```
program example3;

  procedure p8 (i : integer; j : integer); begin end;

  procedure p9 (procedure q3 (i,j : integer)); begin q3(42, 42) end;

begin p9(p8) end.
```

Here the formal parameters of "p8" and "q3" are not congruent because "p8" has two sections in its formal-parameter-list (each of which contains one integer), and "q3" has only one section (which contains two integers). Thus the actual parameter "p8" does not match the formal parameter "q3".

Although only procedural parameters have been discussed, ISO-Pascal imposes similar restrictions on the matching of actual and formal functional parameters. In both these cases a great improvement has been achieved over Wirth's original language; the number, type and sort of parameters is now significant. Unfortunately, the names of parameters must be supplied, although this information is superfluous to congruence and therefore irrelevant. Parameter sections are also required to match (see "example3") and this leads to (unwanted) pedantry from the compiler.

5 Separate Compilation

The ISO standard does not define any facilities for separate compilation in Pascal. There are, however, certain "hints" contained in the BSI specification; program parameters, for example, may be variables of any type, although it is only the behaviour of file variables that is defined. The directive **forward** is the only one that is defined, although **external** is discussed. Perhaps separate compilation can be facilitated using these "hooks"?

The answer is, predictably, no. There is no obvious way to compile types separately, nor is it clear what is meant by them. However Flex-Pascal does support separate compilation [Taylor:86] which provides the same strong type-checking across separately-compiled units as it does within one unit. Many other implementations also provide such a facility, but no two are likely to be the same. The ANSI candidate extension proposal for separate compilation [Pascal 2:84] would provide some much needed standardization in this commonly-used area, although the proposal seems to be providing thinly-disguised Ada library packages.

6 External Files

Flex-Pascal files are handled in an object-oriented manner, being passed as parameters to Pascal programs and system utilities which act upon them [Hammond:85]. In the Flex environment under which programs are compiled and run, objects have definite modes and the command interpreter [Currie:82], is used to manipulate these objects. Thus, from a Pascal program with one parameter of type text, a procedure is derived which has mode (Textfile->Textfile). This procedure can be applied to an actual Textfile using the command interpreter. The Textfile value delivered is simply a copy of the parameter to facilitate further procedure calls.

Files exist in two distinct states: disc-files and mainstore-files. A disc-file is a permanent value which can be kept in the Flex filestore. A mainstore-file is a temporary value and is the form of a file that is passed to a Pascal program. System utilities can be used to convert a file between the two forms.

Disc-files can never be altered, only new disc-files created. Applying a Pascal program to a mainstore-file can change the mainstore-file but not the disc-file from which it was generated. If the present contents of the file are to be kept then a new disc-file must be created with the appropriate system utility. A disc-file is like a frozen copy of a mainstore-file.

Blank or empty files are created by a system utility. In order to generate a mainstore-file with the correct mode the Pascal type of the file must be supplied. The utility takes an editable file which defines the file type using Pascal constant and type definitions.

7 Comments On Possible Extensions to ANSI-Pascal

This section offers comments on extensions proposed in the candidate extension library [Pascal 1:84] produced by the Joint ANSI/IEEE Pascal Standard Commitee, for the ANSI/IEEE standard for Pascal (ANSI/IEEE 770X3.97-1983). The ANSI standard is based extensively on ISO 7185, but omits the conformant array schema. The library of candidate extensions is not final and items in it are not guaranteed to be included in any new standard. Also offered are comments on proposals for additions to this library [Pascal 2:84].

Most of the candidate extensions are easy to implement in the Flex-Pascal compiler (e.g. underscore characters in identifiers) and so are not discussed further. However a few of the extensions raise some interesting points with regard to the changes that would have to be made in the Flex-Pascal compiler in order to implement them.

Constant Expressions

This extension would allow the right-hand side of a constant declaration to be an expression, whereas now it can only be a constant denotation or a constant identifier.

The extension specifies that the expression must have an unambiguous base type, and could not depend on run-time results or user-defined functions. However this still allows an expression such as that for "sin60" in the fragment of program below:

```
...
const sin60 = sin(60);
var x, angle : real;
begin
...
  angle := 60;
  x := sin(angle)
...
end
```

The call of the sine routine in the expression for "sin60" would be evaluated at compile time, whereas the call of the sine routine in a program, procedure or function block, such as the value for "x", would be evaluated at run time. One would expect the same result to be obtained for the same parameter in both cases, which means that the sine routine called in the compiler must be the same as that used in the code produced for the program.

A similar problem may arise if cross-compiling, that is compiling on one machine (the host), producing code to run a different machine (the target). Then if we have

```
const a = 16.234;
      b = a + 5.223;
```

then the expression given for b would again be evaluated at compile time. However it may be that the accuracy of the arithmetic of the host machine is different to that of the target machine, or that the host machine has no real arithmetic at all. The same reals as on the target machine would then have to be implemented on the host machine.

Random Access Files

This extension would add a new type of file:

```
random-access-file = "file" "[" index-type "]" "of" component-type.
```

with associated procedures so that access could be gained to a particular element of the file without having to access all elements prior to it as in the sequential files in the standard at the moment.

As the number of elements in the file cannot exceed the cardinality of the index-type, there is a fixed maximum size for each random access file. This raises a problem similar to one in the area of sets: if one declares a very large file, does the program always consume this amount of space at the start of the program, or only increase it as needed (as elements are put into the file) and maybe run out of store in the middle of the program.

Out of the proposals in the foreword to the work in progress, comments on separate compilation have already been made. One other is:

Open/Associate and Close for Files

These extensions would allow associating a Pascal file with a specific external data set, terminal, device, etc. and to specify certain properties of that file. These seem to be too dependent on the file system of the machine (e.g. by use of concepts such as block sizes) to be in any way 'standard'.

8 Summary

In the previous sections much emphasis has been placed on those aspects of the ISO standard which are ambiguous or could be improved. However it must be stressed that the ISO standard was beneficial to writing the Flex-Pascal compiler; it gave a definite basis from which to work. It should also help to produce more portable programs as processors complying with the standard should compile and run any ISO-Pascal program. The use of Ten15 as an intermediate language was also a great help as it meant that no 'cheating' was possible in producing code.

Acknowledgement

The authors would like to thank Dr. J. M. Foster for all his advice and help.

References

[Ada:83] Ada Joint Program Office, "Reference Manual for the Ada Programming Langauage", United States Department of Defense, ANSI/MIL-STD 1815 A, January 1983.

[Algol68:76] van Wijngaarden A. et al., "Revised Report on the Algorithmic Language Algol 68", Springer-Verlag, 1976.

[BSI:82] British Standards Institution, "Specification for the Computer Programming Language Pascal", BS 6192, 1982.

[Currie:81] Currie I.F., Edwards P.W. & Foster J.M., "Flex Firmware", RSRE Report 81009, 1981.

[Currie:82] Currie I.F. & Foster J.M., "Curt : the Command Interpreter Language for Flex", RSRE Memorandum 3522, 1982.

[Currie:83] Currie I.F., Edwards P.W. & Foster J.M., "Kernel and System Procedures in Flex", RSRE Memorandum 3626, 1983.

[Foster:68] Foster J.M., "A Syntax Improving Program", Computer Journal, VOL II, pp 31, 1968.

[Foster:82] Foster J.M., Currie I.F. & Edwards P.W., "Flex: A Working Computer with an Architecture Based on Procedure Values", RSRE Memorandum 3500, 1982.

[Goodenough:85] Goodenough S.J., Taylor P.D. & Whitaker G.D., "A Guide to SID for Users of the Flex Computer", RSRE Memorandum 3768, 1985.

[Hammond:85] Hammond P.D., "A Procedural Implementation of Pascal Files on Flex", RSRE Memorandum 3883, 1985.

[Pascal 1:84] "Pascal : Foreword to the Candidate Extension Library", ACM SIGPLAN Notices, vol 19, N$^{\underline{o}}$ 7, July 1984.

[Pascal 2:84] "Pascal : Foreword to Work in Progress", ACM SIGPLAN Notices, vol 19, N$^{\underline{o}}$ 7, July 1984.

[Taylor:86] Taylor P.D., "A Separate Compilation System for Flex-Pascal", RSRE Memorandum 3918, 1986.

[Wirth:71] Wirth, N., "The Programming Language Pascal", Acta Informatica, vol 1, fasc. 1, pp 35, 1971.

[Woodward:83] Woodward P.M. & Bond S.G., "Guide to Algol 68 (for users of RS Systems)", Edward Arnold, 1983.

Appendix - The Implementation of Sets

```
DECS sets_m:

MODE SET = REF [] BOOL;

OP MIN = (INT a,b) INT:
IF a > b THEN b ELSE a FI;

OP MAX = (INT a,b) INT:
IF a < b THEN b ELSE a FI;

INT min_int = most negative integer,
    max_int = most positive integer;

PROC union = (SET a,b) SET:
(INT upb_a = UPB a, upb_b = UPB b, lwb_b = LWB b, lwb_a = LWB a;
 IF upb_a < lwb_a
 THEN b
 ELIF upb_b < lwb_b
 THEN a
 ELSE HEAP [lwb_a MIN lwb_b:upb_a MAX upb_b] BOOL c;
      FORALL ci IN c DO ci := FALSE OD;
      c[lwb_a:upb_a AT lwb_a] := a;
      FORALL bi IN b, ci IN c[lwb_b : upb_b AT lwb_b]
      DO ci := ci OREL bi OD;
      c
 FI
);

PROC difference = (SET a,b) SET:
(INT upb_a = UPB a, upb_b = UPB b, lwb_b = LWB b, lwb_a = LWB a,
     upper = upb_a MIN upb_b, lower = lwb_a MAX lwb_b;
 IF lower > upper
 THEN a
 ELSE INT first := lwb_a-1, last := lwb_a-1;
      HEAP [lwb_a : upb_a] BOOL c := a;
      FORALL bi IN b[lower:upper], ci IN c[lower:upper]
      DO ci := ci ANDTH NOT bi OD;
      FOR i FROM lwb_a TO upb_a
      WHILE NOT c[i] OREL (first := i; FALSE)
      DO SKIP OD;
      IF first < lwb_a
      THEN HEAP [1:0] BOOL
      ELSE FOR i FROM upb_a BY -1 TO first
           WHILE NOT c[i] OREL (last := i; FALSE)
           DO SKIP OD;
           c[first:last AT first]
      FI
 FI
);

PROC intersection = (SET a,b) SET:
(INT upb_a = UPB a, upb_b = UPB b, lwb_b = LWB b, lwb_a = LWB a,
     upper = upb_a MIN upb_b, lower = lwb_a MAX lwb_b;
 IF lower > upper
 THEN HEAP [1:0] BOOL
 ELSE INT first := upper, last := lower-1;
      HEAP [lower:upper] BOOL c;
      FOR i FROM lower TO upper
      DO c[i] := a[i] ANDTH b[i] ANDTH (first := first MIN i;last := i; TRUE)
      OD;
      IF last < first
      THEN HEAP [1:0] BOOL
      ELSE c[first:last AT first]
      FI
 FI
);

PROC equality = (SET a,b) BOOL:
(LWB a = LWB b) ANDTH (UPB a = UPB b) ANDTH
   (BOOL eq := TRUE;
      FORALL ai IN a, bi IN b
      WHILE ai = bi OREL (eq := FALSE; FALSE)
      DO SKIP OD;
      eq
     );

PROC inequality = (SET a,b) BOOL:
(LWB a /= LWB b) OREL (UPB a /= UPB b) OREL
    (BOOL neq := FALSE;
     FORALL ai IN a, bi IN b
     WHILE ai = bi OREL (neq := TRUE; FALSE)
     DO SKIP OD;
     neq
    );

PROC contains = (SET a,b) BOOL:
(INT upb_a = UPB a, upb_b = UPB b, lwb_b = LWB b, lwb_a = LWB a;
 lwb_b >= lwb_a ANDTH upb_b <= upb_a ANDTH
    (BOOL c := TRUE;
     FORALL ai IN a[lwb_b:upb_b AT lwb_b], bi IN b
     WHILE bi ANDTH (ai OREL (c := FALSE; FALSE)); c
     DO SKIP OD;
     c
    )
);

PROC is_contained = (SET a,b) BOOL:
(INT upb_a = UPB a, upb_b = UPB b, lwb_b = LWB b, lwb_a = LWB a;
 lwb_a >= lwb_b ANDTH upb_a <= upb_b ANDTH
    (BOOL c := TRUE;
     FORALL ai IN a, bi IN b[lwb_a:upb_a AT lwb_a]
     WHILE ai ANDTH (bi OREL (c := FALSE; FALSE)); c
     DO SKIP OD;
     c
    )
);

PROC in = (INT i, SET s) BOOL: i >= LWB s ANDTH i <= UPB s ANDTH s[i];

PROC build = (VECTOR [] INT r,s) SET:
(INT upper := min_int, lower := max_int;
 FOR i BY 2 TO UPB r
 DO IF r[i+1] >= r[i]
    THEN upper := upper MAX r[i+1];
         lower := lower MIN r[i]
    FI
 OD;

 FORALL si IN s
 DO upper := upper MAX si; lower := lower MIN si OD;

 IF upper < lower
 THEN HEAP [1:0] BOOL
 ELSE HEAP [lower:upper] BOOL set;
      FORALL set_i IN set DO set_i := FALSE OD;
      set
 FI
);

PROC constructor = (VECTOR [] INT r,s) SET:
(SET set = build(r,s);
 FOR i BY 2 TO UPB r DO FOR j FROM r[i] TO r[i+1] DO set[j] := TRUE OD OD;
 FORALL si IN s DO set[si] := TRUE OD;
 set
)

KEEP union,difference,intersection,equality,inequality,contains,is_contained,
     in,constructor,SET
FINISH
```

DOCUMENT CONTROL SHEET

Overall security classification of sheet Unclassified

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference Memorandum 3908	3. Agency Reference	4. Report Security Classification U/C
5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location Royal Signals and Radar Establishment		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location		

7. Title
FLEX Pascal: an implementation of the 150-Pascal programming language

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials Curtis K	9(a) Author 2 Hammond PD	9(b) Authors 3,4... Taylor PD	10. Date	pp. ref.
11. Contract Number	12. Period	13. Project	14. Other Reference	

15. Distribution statement Unlimited

Descriptors (or keywords)

continue on separate piece of paper

Abstract

The Flex-Pascal compiler runs on the RSRE Flex computor architecture, which is currently available on the ICL Perq 2 workstation. Flex-Pascal conforms to the ISO 7185 (BS 6192) standard for Pascal and provides (as one extension) a separate-compilation facility. This paper discusses the use of ISO 7185 and the BSI Pascal Validation Suite, and criticises areas of Pascal which, by their nature, have proved particularly difficult to implement.
This memorandum is for advance information. It is not necessarily to be regarded as a final or official statement by Procurement Executive, Ministry of Defence.

S80/48