Pascal Compiler Validation

Pascal Compiler Validation

Pascal Compiler Validation

Edited by

Brian A. Wichmann

and

Z. J. Ciechanowicz

Programming Language Standards Group,
National Physical Laboratory,
Teddington.

JOHN WILEY & SONS
Chichester · New York · Brisbane · Toronto · Singapore

Library of Congress Cataloging in Publication Data:

Wichmann, Brian A.
Pascal compiler validation.
Includes index.
1. PASCAL (Computer program language)
2. Compiling (Electronic computers) 3. Computer programs—Validation.
I. Title
QA76.73.P2W53 1983 001.64'25 82-23882
ISBN 0 471 90133 4

British Library Cataloguing in Publication Data:

Pascal compiler validation.
1. PASCAL (Computer program language)
I. Wichmann, Brian A.
001.6'4'24 QA76.6.P2
ISBN 0 471 90133 4

Printed in Great Britain

Acknowledgements

The Editors would like to thank the contributors to this volume for providing the revised text for publication. Owing to the incompatibilities between word processing systems, most of the text was retyped at NPL. The large task that this involved together with a preliminary transcription of the discussion by the typists at NPL made this volume possible. A version of the layout program Pages was used to prepare the masters for printing, and the assistance of C W Nott in this respect was most helpful.

List of Contributors

A M Addyman, Chairman OIS/5 (British Standards Institution Committee on programming languages), Chairman, ISO Working Group on Pascal (TC97/SC5/WG4), Lecturer at University of Manchester, Manchester M13 9PL, UK.

B Byrne, Manager, ICL Pascal programming support group, ICL, Lovelace Road, Bracknell, Berkshire, RG12 4SN, UK.

J Charter, Assistant Director, Certification and Assessment Services, British Standards Institution, Hemel Hempstead, Herts HP2 4SQ, UK.

Z J Ciechanowicz, National Physical Laboratory, Teddington, Middlesex, TW11 OLW, UK.

M Gien, Director, Project SOL, Institut National de Recherche en Informatique et Automatique, 78153 Le Chesnay, France.

A Hay, Lecturer, University of Manchester Institute of Science and Technology, Manchester, M60 1QD, UK.

C C Kirkham, Lecturer, University of Manchester, Manchester M13 9PL, UK.

L Morgan, Head, Programming Language Standards Group, National Computing Centre, Oxford Road, Manchester, M1 7ED, UK.

A H J Sale, Professor of Computer Science, University of Tasmania, Hobart, Tasmania, Australia.

R S Scowen, National Physical Laboratory, Teddington, Middlesex, TW11 OLW, UK.

Jacqueline Sidi, Bureau d'orientation de la Normalisation en Informatique, 78153 Le Chesnay, France.

K Thompson, Directorate III/B, Commission of the European Communities, Rue de la Loi 200, B-1049, Brussels, Belgium. *(Editorial note: It was not possible to include his paper in these proceedings; however, he has contributed to the discussion.)*

J Welsh, Professor of Computer Science, University of Manchester Institute of Science and Technology, Manchester, M60 1QD, UK.

B A Wichmann, Head, Programming Language Standards Group, National Physical Laboratory, Teddington, Middlesex, TW11 OLW, UK.

Contents

Foreword xi

1. **Introduction** 1
   B. A. Wichmann

2. **The Pascal compiler validation project** 4
   B. A. Wichmann

3. **The Pascal Standard** 6
   A. M. Addyman and C. C. Kirkham

4. **The validation suite** 15
   A. H. J. Sale and B. A. Wichmann

5. **Developing the testing procedures** 26
   B. A. Wichmann and Z. J. Ciechanowicz

6. **Second thoughts on the validation suite** 35
   A. H. J. Sale

7. **The Pascal Standard from the implementor's viewpoint** 46
   J. Welsh and A. Hay

8. **A manufacturer's viewpoint** 59
   B. A. Byrne

9. **Pascal validation users' guide** 64
   L. Morgan

10. **The role of BSI in testing** 69
    J. W. Charter

11. **The SOL project and validation** 75
    M. Gien and J. Sidi

12. **Discussion** 81

13. **Compiler validation—a survey** 90
    R. S. Scowen and Z. J. Ciechanowicz

Appendix A Extracts from the Pascal Test Suite 145

Appendix B Has the program been altered? 159
B. A. Wichmann

Appendix C The Assumptions program 169

Index 171

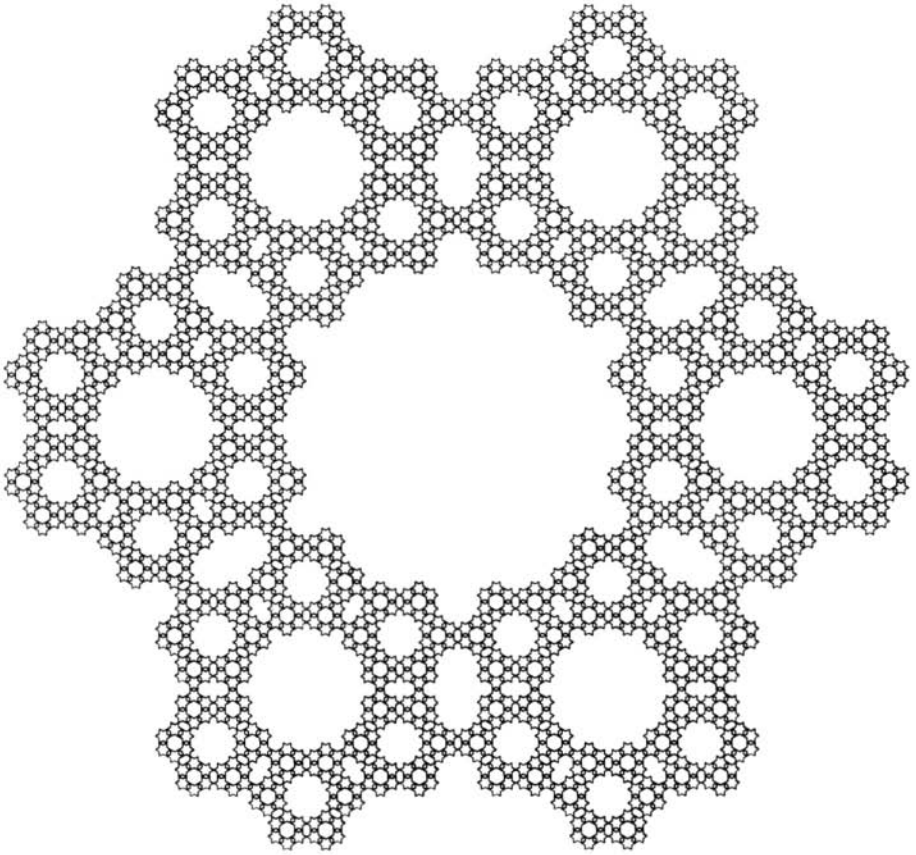
Foreword

When I first contemplated the prospect of proving correctness of programs, I knew how useful and important this would be; but I never thought I would be able to do it myself. When I first tried it, I found it so difficult that I was tempted to test the program first, to check that I was not wasting my efforts on the fool's errand of trying to prove something false.

Now the verification of small programs is set as an exercise for students; but the verification by proof of a programming language implementation is still a daunting task. The first prerequisite is a clear and complete specification of the language itself. This has already been accomplished for Pascal by the painstaking efforts of standardization bodies. The assembly of a comprehensive suite of test programs is obviously the next step; and thanks to the hard work of the contributors to this volume, that task has also been accomplished. Their pioneering work requires study by all intending implementors of Pascal, and by all serious users of the new Standard.

But we must not lose sight of the ultimate goal, that of the design and construction of programming language implementations which are known to be correct by virtue of the quality of the logical reasoning which has been devoted to them. Of course, such an implementation will still need to be comprehensively tested before delivery; but it will immediately pass all the tests, and then continue to work correctly for the benefit of programmers forever after.

C A R Hoare, FRS
Professor of Computation
Programming Research Group
Oxford



Hexagons within hexagons within hexagons and so on *ad infinitum*. Recursion allows a natural expression of the repeating patterns in nature. Simply programmed in Pascal this is an advanced exercise in FORTRAN.

1

Introduction

B. A. Wichmann

Over the last few years, the programming language Pascal has grown in popularity greatly. It is widely used for teaching in Universities, is available on most micro-processors and main-frames as well. In fact, Pascal is one of the few languages that form a bridge between micro-processor systems and the main-frame world.

Until recently, there has been one drawback to Pascal as a general purpose software tool. The definition of the language was not very precise and in consequence, the portability of Pascal programs was problematic. The British Standards Institution(BSI) set up a group under Dr Tony Addyman to produce a standard definition of the language. This was later superseded by an ISO group also under Tony Addyman. In October 1981, ISO agreed to the standardization of Pascal, and after editorial work on the document, BSI published the Standard in February 1982 (BS 6192).

What does this mean for users of Pascal? The portability of Pascal programs should be much improved provided suppliers implement the Standard and users write their programs to conform to the Standard. One might think that the position with Pascal is no different from that of COBOL or FORTRAN and yet portability problems arise with these languages. There are several reasons for believing that Pascal is different:

1. The Pascal Standard is more comprehensive than that of COBOL or FORTRAN. For instance, the COBOL and FORTRAN standards do not require that an invalid program is rejected by a compiler. The Standard for these languages is just a definition of a language rather than a set of requirements for a compiler. This is clearly not very satisfactory since we all write incorrect programs on occasions.

2. The Pascal Standard is simple and devoid of a multitude of options. If the language has lots of options, then program portability is reduced because a program may not be valid without a specific option. COBOL has a large number of options and FORTRAN 77 has two major levels (essentially distinct languages) whereas Standard Pascal has just one option, affecting only one part of the language. This option is to allow procedures to handle arrays whose size varies from call to call. This option, level 1 Pascal, would allow Pascal programs to call FORTRAN routines in many systems.

3. The Pascal test suite is more searching than that of COBOL and FORTRAN. This is essentially a consequence of the definition of the language. The National Physical Laboratory has been collaborating with the University of Tasmania on the construction of this suite for over two years. About 400 copies of the test suite have been sold worldwide. A new version of this suite has recently been issued to correspond to the

new ISO Standard. Unlike the COBOL and FORTRAN test suites, the one for Pascal includes incorrect programs which must be rejected: ones to examine the error-handling capability of a compiler, and the "quality" of an implementation. The quality tests indicate if there is any small limit to the complexity of programs that a system can handle and also assesses the accuracy of real arithmetic.

All the major components to make Pascal a good Standard are now available, that is, a Standard definition and tests to verify conformance of a compiler to the Standard.

A Standard and tests to check conformance to the Standard are not alone quite sufficient. The test procedures must be used and results made known to those using Pascal compilers. This can be achieved by independent testing of compilers which is currently being investigated by BSI (Hemel Hempstead). BSI have a wealth of experience with testing other goods but this is their first venture into computer software. For this reason, both NPL and NCC are assisting BSI in this important development.

The last step in this process is to encourage users to request a *Standard* compiler from the suppliers and for suppliers to meet that demand. As a contribution to this last step, NPL held a conference on this topic with its collaborators in February 1982. This book includes the material presented at that conference. In addition, extracts from the Pascal test suite are included together with other supporting material so that the readers may have a full understanding of the validation of Pascal from this publication alone. Also included as the final chapter is a survey on compiler validation as a whole which was undertaken at NPL as part of this project. Although previously available as an NPL report, its inclusion here should make the material more generally accessible.

Finally in this introduction, we illustrate the need for better quality Pascal compilers by a simple example. This example has five deliberate errors which are known to have gone unnoticed by one compiler in the past. You can test your knowledge of Pascal by finding the errors. Then you can compare your success with any compiler by typing it into the system.

The incorrect Pascal program with five errors.

```
program test;
const
   nil = '0';
begin
   if nil # '0' then
      writeln( 'WRONG', +nil, .123)
   else
      writeln( 'RIGHT' )
end.
```

```
Errors
1. The program must contain output as parameter.
2. nil cannot be used as an identifier (it is a reserved word).
3. # is written as <> (not equals).
4. nil cannot follow a sign.
5. a decimal point must follow a digit.
```

The corrected program is:

```
program test(output);
const
   nill = '0';
begin
   if nill <> '0' then
      writeln( 'WRONG', nill, 0.123)
   else
      writeln( 'RIGHT' )
end.
```

Although this test is only an illustration, it does show the wide ranging capabilities of current compilers. The results of compilers tested so far can be summarised thus:

Compiler	Errors detected	Accuracy of error messages	Recovery from last error
A	4	3	4
B	2.5	2	3
C	2	2	2
D	1	2	1
E	2.5	3	2
F	3.5	3	3
G	4.5	4	3
H	5	4	4
I	3.5	1	2

All the marks are out of 5. The half marked for detecting an error show that the error message was confusing enough for it to be unclear if the error was properly detected. The last two columns show how accurately the messages indicated the actual error. The error is often given by a number in which case the text given in the manual was used to assess the result. Naturally, this column involves a subjective judgement. The last column indicates how well the compiler recovered from the error in order to proceed with the accurate detection of further errors. Of course, it is unlikely that the further errors in this test would be located with poor recovery, but compilers often produce many additional messages which can be confusing to the programmer (unless he is also a compiler-writer). This last column is also subjective.

2
The Pascal compiler validation project

B. A. Wichmann

Providing an adequate definition of a programming language is not easy. Although Algol 60 created a new standard for the syntactic definition of a language, the semantics were nevertheless imprecise in many areas. Fortunately, most of the ambiguities in Algol 60 were mainly of concern to academics and did not require resolution for the correct execution of ordinary programs. One method of illustrating an ambiguity is to write a program whose interpretation can vary. This program can then be executed on any available implementation to see what interpretation has been taken. This approach was taken by the author for Algol 60 which resulted in some test reports [1] and eventually led to the revision of the language [2].

It might be thought that successors to Algol 60 would be more precise in their definition and that at least the known trouble-spots in Algol 60 would be avoided. This is not true of the original definition of Pascal given by Wirth [3]. A natural development was therefore to recast the Algol 60 test programs in Pascal and also to write new ones to reflect the new concepts in the language. This revealed in a concrete form many of the known ambiguities in the language.

Not long after the publication of the Pascal report, it became clear that the language was likely to be of major significance. The language is easy to teach, implementable even on microprocessors, and appropriate for a reasonably large class of applications. The enthusiasm of a large part of the academic community for Pascal has been a major reason for the rise in its popularity. Hence there was the need for a more precise and internationally agreed definition. This led to the formation of a BSI group to define Pascal chaired by A M Addyman. After the publication of a draft BSI standard for Pascal, an ISO working group was formed for Pascal also chaired by A M Addyman.

An enthusiastic supporter of Pascal has been A H J Sale who has maintained strong links with the BSI standardization effort from the most distant point possible. He added very substantially to the test programs written at NPL so that a period of very fruitful collaboration was started. It is important to note that this effort ran in parallel with that of defining the Standard. In the author's view this parallel development had a very beneficial effect on both the Standard and the test suite.

An important aspect of the test programs, both for Algol 60 and Pascal, was that anything could be written. In particular, many of the programs were invalid. Indeed, an important result from the Algol tests was that twice as many compiler errors were located from incorrect programs as correct ones. There has been a similar experience with Pascal, although now that the test suite has been so widely used, it is difficult to find a compiler that does not show some evidence of having been corrected in the light of executing the

test suite. As the Standard was refined, the test suite changed from one which illustrated defects in the definition to a test suite for Pascal compilers which implemented the Standard.

As work on the Standard developed, so an attempt was made to keep the test suite in line with it. The main influence here was the classification of the test suite programs, such as conformance tests (i.e. correct Pascal programs which should execute to completion). By the time version 2.2 of the test suite was issued (September 1979), its size and the number of copies distributed were such that making changes was not easy. Of course, with over 300 programs, a rigid discipline must be followed and experience has shown that the standards used for the test programs should largely be checked by program. The gap of over two years before version 3.0 was released (January 1982), was largely caused by the hope that the ISO standard would be issued "shortly".

Over the past 18 months, NPL has been collaborating with both the National Computing Centre and the British Standards Institution on this project. The aim is to enable BSI to set up a formal validation service, not dissimilar from the one currently run by the General Services Agency in the United States. The US service is a government run project whose purpose is limited to checking conformance of COBOL and FORTRAN compilers for government procurement only. However, since the test reports are publicly available, the facility can easily be used for other purposes. The UK effort is slightly different in that BSI (Hemel Hempstead) is a test house, independent of the Government, which tests to a wide range of standards including safety standards such as seat belts. The provision of an ISO and BSI standard for Pascal therefore makes it appropriate for BSI to test for compliance with the Standard. The role of NCC has been to provide support to BSI in this new move by BSI in testing software. NCC have been responsible for producing a guide on the test procedures needed for formal validation.

Although this work was inspired by a desire to understand and define languages better, it should not be thought of in purely academic terms. Program portability is not easily achieved unless compilers are more uniform than they are at the moment (as can be seen from FORTRAN and COBOL). Moreover, the increased use of Pascal on a very wide variety of machines makes the portability of a Pascal program commercially attractive.

This paper has surveyed the history of the project. The following papers relate to the current state of the project, its future and its role in computing generally.

References

[1] B Jones and B A Wichmann, Testing Algol 60 compilers, Software - Practice and Experience, Vol 6, 1976 pp261-270.

[2] R M DeMorgan, I D Hill and B A Wichmann, Modified Report on the Algorithmic language Algol 60. Computer Journal, Vol 19, 1976, pp276-288. Also published with officially agreed corrections by the Dutch standards body for ISO.

[3] N Wirth, The Programming Language Pascal, Acta Informatica, Vol 1, pp35-63, 1971.

3

The Pascal Standard

A. M. Addyman and C. C. Kirkham

Summary

For the benefit of those who have seen one or more of the drafts, or who are interested in the development of the Standard, a brief history of the project is given. This is followed by a description of some of the guiding principles which were applied during this development. As a result of the project the definition of Pascal exhibits changes and clarifications when compared with the Pascal of the User Manual and Report. Several of these changes are examined.

History

In 1977 a working group (designated DPS/13/4) was formed by the BSI technical committee responsible for the standardization of programming languages (then DPS/13, now OIS/5). The working group was given the task of producing the draft for a British Standard for Pascal. This group shouldered the responsibility for the production of the first four working drafts. As a description of the early years appears elsewhere [1], attention will be focussed on the major milestones.

The Attention List

The working group first produced a list of all known problems with the current definition of Pascal. This was called the Attention List. The final version ran to 17 pages. It contained contributions from members of the group, from correspondents and from published criticisms of Pascal. It is interesting to note that this list proved to be far from complete. Many of the really nasty problems did not surface until later. In April 1978 it was decided that further work on the Attention List should be suspended in favour of the production of a draft.

The First Working Draft

The first working draft was produced by dividing up the subject matter into some 16 sections. Many of the section topics corresponded to those of the Revised Report. Each section was the responsibility of two group members; one to write the section and one to comment upon it. The first draft was completed by mid-July 1978.

The Second Working Draft (The 'San Diego' Draft)

Immediately following receipt of all the sections which formed the first working draft, the convenor (A.M.Addyman) was a participant at a workshop which had been organised to discuss Pascal. The first working draft was the subject of informal discussions at the workshop. On returning from the workshop, the document was redrafted to: (a) remove the obvious inconsistencies in style etc. (b) incorporate comments from the informal discussions. Both the first and second working drafts were distributed to the members of the group during August 1978.

The Third Working Draft

A meeting of the group was held in September 1978 to discuss the drafts. This resulted in a list of corrections and amendments to the second working draft. These were incorporated in a new draft [2,3].

The Draft For Public Comment

The draft for public comment was an edited version of the third working draft. The changes introduced brought the document into line with BSI/ISO editorial practice. This document became BSI document 79/60528 DC and ISO/TC 97/SC 5 N462.

The Fourth Working Draft

The fourth working draft was produced in response to the comments received by August 1979. The group (DPS/13/4) met in September 1979 to discuss these comments and create the draft. This document was circulated as ISO/TC 97/SC 5 N510. It was further discussed at a November meeting.

The Fifth Working Draft

This draft was the result of an ISO Pascal Experts Group meeting in Turin, Italy in November 1979. It was circulated to members of the group in December 1979 for editorial and typographical comments. It was registered by the ANSI committee as X3J9/80-003.

The First ISO Draft Proposal (ISO/TC 97/SC 5 N565)

This document was produced as a result of the comments received from the Experts Group. It also appeared in [4]. The SC5 letter ballot on DP7185 produced nine positive and four negative votes.

The Second ISO Draft Proposal (ISO/TC 97/SC 5 N595)

The second DP was produced after a long delay which was caused by the disagreement over the Conformant Array feature. The ISO working group (ISO/TC 97/SC 5/WG 4) which met to discuss the comments received on N565

were unable to resolve the Conformant Array issue at the meeting. The issue was resolved by correspondence (after a meeting of the UK group and the intervention of certain eminent persons). The SC5 letter ballot produced thirteen positive and one negative vote. The negative vote was cast by Japan who voted against *starting* the project as well as against both DPs. Of the six P-members who failed to vote, two had approved on the previous ballot.

The Pascal Standard (BS 6192)

At its meeting in October 1981, SC5 agreed to the registration of DP7185 as a Draft International Standard (i.e it is ISO/DIS 7185) and agreed that the English text of the DIS would simply be a reference to BS 6192. A copy of the agreed text was circulated, for information, as the Third Draft Proposal (ISO/TC 97/SC 5 N678). The final ballot within ISO to confirm the DIS as an International Standard has been waiting for publication of BS 6192 [5,6]; the French translation was completed before the October meeting and as a consequence the final ballot will not be delayed for lack of the translation. This is a tribute to the work of the French Pascal experts as a delay of one or two years at this stage is not uncommon.

The guiding principles

Early in the project a number of guiding principles evolved which made the standardization of Pascal very different from the standardization of other programming languages. From the start, it was the intention of those involved to *standardize* Pascal not to *change* Pascal. This is not the case with other languages e.g. FORTRAN and COBOL where change, in the guise of improvement, is very much the order of the day. So much so, that it has been said of FORTRAN that "FORTRAN is the name of the language that X3J3 standardizes"! As a consequence of this principle, the emphasis during the standardization of Pascal has been on the clarification of successive drafts. Even so, some language changes were introduced as a result of the comments received from ISO member bodies etc. These changes will be described later. The intention insofar as the clarifications were concerned was, of course, to get them right. This was not always a peaceful process because one person's clarification is another person's change! To be able to judge what is right implies the existence of some criteria which may be used. Although they were never written down, or even formally discussed, suitable guidelines did evolve. Examples of these guidelines are:

1. We should not take a decision which was biased against a particular implementation strategy. This guideline affected the Conformant Array issue and the clauses dealing with scope.

2. We should do everything possible to enhance the security of the language. In particular, we have a preference for semantic rules which can be checked by inspecting the program text, i.e. at compile time. There are numerous examples of the many aspects of this guideline – restrictions on tagfield usage were introduced to facilitate run-time error detection; the semantics of the with-statement were clarified to eliminate a possible error situation; the for-statement and goto-statement were made completely compile-time checkable.

The major changes

There are two major changes to Pascal which have been brought about by the Standard, both are in the area of procedure parameters. The first, which was the use of an embedded procedure heading to specify the parameters of a parametric procedure, was introduced in the third working draft. This change was requested by Prof. Wirth. The second change, the introduction of a special form of array parameter, proved to be somewhat controversial. The decision to make such a change was taken by DPS/13/4 while processing the comments received on N462. This decision was endorsed by the ISO Experts in Turin and supported by the vast majority of the member bodies of SC5. Although many of the member bodies requested (even demanded) enhancements to the proposal, only two actually opposed the change. The reasons given for their opposition are interesting. They were: (a) it was a significant extension to Pascal; (b) a language designer must not add to a language any feature that is not very well understood, that has not been implemented, or that has not been used in real programs; (c) if the Standard was defining the language for the sake of portability, then conformant arrays should be omitted; (d) other important extensions are needed also. It should be noted that both Professors Wirth and Hoare, who have some responsibility for the design of Pascal, expressed the view that a standard for Pascal should permit the passing of array parameters of varying size.

Conformant Arrays

The conformant array parameter evolved over the period from Autumn 1979 to Summer 1981. This evolution can be traced through the various drafts beginning with N510.

ISO/TC 97/SC 5 N510

The proposal included in N510 was universally disliked and was abandoned in favour of a proposal drafted by Prof Sale, that was itself based on the 'bound-spec' proposal suggested by Prof Wirth as an extension to the Zurich compiler after responsibility for that compiler had been transferred to the University of Minnesota.

ISO/TC 97/SC 5 N565

The syntax was essentially that of the final version except that neither packed conformant arrays nor value conformant arrays were permitted. These restrictions were incorporated because (a) the introduction of value conformant arrays would mean that the size of procedure activation records could not always be determined at compile-time; (b) the interaction between packed multi-dimensional arrays and conformant arrays would cause severe implementation problems. This proposal was criticised because of these restrictions. In August 1980 a revised proposal was circulated. This proposal had the following characteristics.

1. The Pascal string types were changed to be either packed or unpacked arrays.

2. Value parameters could also be conformant arrays, but the corresponding actual parameters were subject to certain restrictions.

These restrictions ensured that the copying of an array which was the actual parameter of a conformant array parameter was only necessary when the size of the array could be determined at compile time. The copy was to be made by the CALLING procedure and a reference to it passed as the parameter. The restrictions were quite complex and involved making the array 'read-only' within the called procedure.

ISO/TC 97/SC 5 N595

In October 1980 a proposal was circulated which became the one incorporated in the second DP. The essence of the proposal was:

1. The conformant array parameter feature became optional. This was achieved by creating two levels of compliance for both programs and processors. Level 0 excluded the feature; level 1 included it. The USA had proposed that (as a compromise) the feature should be made optional. The actual wording used incorporated certain improvements over the original US suggestion.

2. The definition of a string type was unchanged.

3. The syntax was altered to permit the specification of a conformant array which had its innermost (or only) dimension packed. This allowed the use of strings as actual parameters.

4. In the previous proposal, conformant arrays could be used both as var and value parameters. However, the implementation of value conformant arrays, which the proposal implied, involved the CALLING procedure copying the array and passing a reference to it to the called procedure. This was different from the usual implementation of value parameters. In fact, it produced an effect similar to that which a programmer could produce by using an additional variable in the calling procedure, and explicitly copying the array prior to using a var conformant array (which need not be read-only!). This observation formed a basis for the proposal. All conformant array parameters were passed as a reference to a variable. If the actual parameter were not a variable, the value was assigned to a suitable anonymous variable in the CALLING procedure. The read-only restriction was no longer necessary, but similar checks could be used in an optimising compiler to avoid the copy operation. This proposal was simpler and more honest than its predecessor. Confidence in its implementability was strengthened because this parameter passing mechanism is to be found in PL/I and the implementation technique is very common in Fortran compilers.

5. The choice of a fifth kind of parameter rather than variations on two existing kinds of parameter facilitated the drafting of a standard incorporating the conformant array parameter as an optional feature.

BS 6192

As a result of the comments from ISO member bodies, the syntax has been changed so that the parameter passing semantics are determined by the form of the parameter specification NOT by the form of the actual parameter. Otherwise the details are unchanged from N595.

Examples

```
procedure AddVectors (var A, B, C:
        array [low..high: natural] of real);
var i : natural;

begin
  for i := low to high do A[i] := B[i] + C[i]
end { of AddVectors };
```

```
function Length (var S: packed array
                [lower..higher: natural] of char)
                            : natural;

begin
Length := upper - lower + 1
end { of Length };
```

Major clarifications

There have been so many clarifications during the development of the Standard that it is difficult to be selective.

The Type Rules

These cover an extraordinary omission from the User Manual and Report. The Standard has two separate concepts - the concept that two objects possess the same type and the concept of assignment-compatibility (which determines whether or not a particular value is valid in a given context). These ideas were developed at an early stage and have proved to be remarkably non-controversial, although the wording has shown a gradual improvement. Comments on N462 resulted in the idea of 'identical types' being replaced by the idea of 'the same type'. Comments on N565 caused the distinction between the definition of a type and the use of a type to be reflected by the syntax rules.

The Scope Rules

The scope rules have been the subject of some debate, the results of which may be considered surprising. The evolutionary process is quite interesting. The third working draft contained (a) the concept of a defining

occurrence (b) declaration before use (c) a specification of the range associated with each defining occurrence (d) ranges which included program text *before* the defining occurrence in which the identifier could not be used. This formulation was criticised for several reasons (a) the term 'range' (in the Algol 68 sense) was not widely known (b) a parameter identifier only denoted a parameter! These and other comments resulted in an improved formulation (see N565) in which (a) the term *defining occurrence* became *defining point* because there is not always an occurrence, and *range* became *region*. (b) identifiers defined in a formal parameter list had two regions! In the formal parameter list the identifier is a parameter-identifier; in the associated block, if any, it is a variable-identifier. Of these clarifications, only the last one caused any serious subsequent debate. Given the nature of the debate and its result, it is interesting to discover why this formulation was chosen. The formulation was necessary because nested procedure-headings could be used to specify procedural parameters and such headings had no block, and consequently no variables. Also, the use of a forward declaration could cause the formal parameter list and the block to be physically separated, so it seemed appropriate for them to be separate regions. The criticisms of this centred on the use of two regions. The objectors disliked the fact that the scope of a parameter and the scope of a declared variable were different. They wanted the scope of all identifiers declared in a procedure to include both the formal parameter list and the block. This criticism had also been made in the comments on N462. These comments were fortuitously answered by the work of Jim Miner [7], who discovered that the published algorithm for checking the uniqueness-of-association rule [8] relied on the existence of these two regions. So, the formulation of the scope rules of Pascal was affected by the seemingly unrelated topics of parametric procedures and forward declarations. Another view of this topic can be found in [9].

Interactive Input/Output

This topic was the subject of considerable comment and several proposals for language changes. These proposed changes were of little use since they considered an interaction to be on a line by line basis, when the real difficulty was posed by single character interactions. Early in the project the implementation technique known as 'lazy I/O' was published [10] and this was adopted as the solution to the problem. The Standard has been criticised for not *requiring* the implementation of lazy I/O, but such a requirement would be unrealistic since interactive input/output relies on many other factors, e.g. the operating system, the terminal etc all of which are beyond the scope of the Standard and which significantly affect the portability of interactive programs. The Standard does, however, require the implementor to document the way in which his processor handles interactive I/O. Users should note that this is a part of the standard which is not currently investigated by the Validation Suite, so that a processor may perform well with the Suite and yet violate the Standard by, for example, changing the semantics of the procedure *read* when it is applied to the file *input*.

Violations and Errors

One of the concerns early in the project was to identify all the places in the specification where an implementor would need to check a semantic

rule during program execution. To distinguish these from other violations, which could always be detected by inspection of the program text, the term *error* was introduced. Originally, it was intended that detection of such errors would be mandatory, but it was argued [11] that this placed unreasonable constraints on the implementor who was responsible for decisions relating to the *quality* of his processor. Issues such as the amount of error detection performed, the efficiency of the generated code, the accuracy of the real arithmetic etc should all be determined by the implementor who knows the demands of his intended market-place. To help prospective users of a processor to judge its error detecting capabilities and to encourage the implementors to detect as many errors as possible, an implementor is required to provide a specific piece of documentation which identifies all the errors which his processor CANNOT detect. The definition of *error* has changed to emphasize the fact that detection is optional rather than the fact that detection usually requires execution of the program. In many ways this change of definition is unfortunate. It was introduced at the suggestion of a group whose intentions were the same as the authors of the original wording, yet the change has succeeded in blurring a distinction that was clear-cut. As a result, attempts were made to alter the classification of certain semantic rules so that many run-time checks would have become mandatory while the detection of a violation of the variant-part completeness rule would have been optional! They did not succeed. Finally, mention should be made of Fischer and LeBlanc [12], whose work on the implementation of run-time checks influenced the wording of the Standard in several areas, particularly those relating to procedure parameters and to identified-variables.

The future

Although we now have a standard for Pascal, important work remains to be done. A minor part of this work relates to the possible production of some ISO technical reports. The major part, indeed the most important part, is to be found in the application of the standard. Users should write programs which comply with it and insist on compilers which comply also. Teachers should ensure that their students are taught the correct language, and not some language that is merely called Pascal! Implementors should ensure that their processors comply with the Standard. To help us in this we have a priceless tool - The Pascal Validation Suite. This is where our energies should now be directed.

References

[1] I.D.Hill and B.L.Meek, 'Programming Language Standardisation', Ellis Horwood, 1980.

[2] A.M.Addyman, Pascal News number 14 - Special issue on the Pascal Standard incorporating WD/3, January 1979.

[3] A.M.Addyman et al, 'A Draft Description of Pascal', Software - Practice and Experience, Vol 9, No. 5 (May 1979).

[4] A.M.Addyman, 'A Draft Proposal for Pascal', SIGPLAN Notices, Vol 15, No.4, (April 1980).

[5] BS 6192 : Specification for Computer programming language Pascal, British Standards Institution, 1982, ISBN 0 580 12531 9.

[6] I.R.Wilson and A.M.Addyman, 'A Practical Introduction to Pascal with BS 6192', The Macmillan Press, 1982.

[7] J.F.Miner, Private communication, 20/10/79.

[8] A.H.J.Sale, 'Scope and One-Pass Compilers', Australian Computer Science Communications, Vol 1 No. 2, (April 1979). (Reprinted in Pascal News No. 15).

[9] A.H.J.Sale, 'Forward-declared Procedures, Parameter-lists and Scope', Software - Practice and Experience, Vol 11 No.2 (February 1981).

[10] J.B.Sare and A.Hisgen, 'Lazy Evaluation of the File Buffer for Interactive I/O', Pascal News No.13 (December 1979).

[11] C.C.Kirkham, Private communication.

[12] C.N.Fischer and R.J.LeBlanc, 'Efficient Implementation and Optimization of Run-Time Checking in Pascal', SIGPLAN Notices, Vol 12 No.3 (March 1977).

4

The validation suite

A. H. J. Sale and B. A. Wichmann

1. Introduction and purpose

This paper describes a suite of test programs which has been designed to support the Standard [1] for the programming language Pascal [6]. It therefore follows similar work done by AFSC [2] for COBOL, and for Algol 60 [3, 4, 5].

The suite of programs is called a validation suite for Pascal processors; however it is important to emphasize that no amount of testing can ensure that a processor that passes all tests is error-free. Inherent in each test are some assumptions about possible processors and their designs; a processor which violates an assumption may apparently pass the test without doing so in reality. Also, some violations may simply not be tested because they never occurred to the validation suite designers, nor were generated from the Standard.

Two examples may illustrate this as a warning to users against expecting too much. Firstly, consider a fully interpretive Pascal processor. It may pass a test which contains a declaration which it would mis-handle otherwise, simply because the program did not include an access to the object concerned so that it was never interpreted. A second example might be a Pascal processor which employs a transformation of the Pascal syntax rules. Since the pathological cases incorporated into the test programs are based on the original rules, a mistake in transformation may not be detected by the test programs.

On the other hand, the test suite contains a large number of test cases which exercise a Pascal processor fairly thoroughly. Hence passing the tests is a strong indication that the processor is well-designed and unlikely to give trouble in use. The validation suite may therefore be of interest to two main groups: implementors of Pascal, and users of Pascal.

Implementors of Pascal may use the test series to assist them in producing an error-free processor. The recommended practice is to use the tests only after conventional testing has been completed, otherwise there is a higher risk that bugs will remain in the processor. The large number of tests, and their independent origin, will assist in detecting many probable implementation errors. The suite may also be of use for re-validating a processor after incorporating a new feature, or correcting an error.

Users of Pascal, which includes actual programming users, users of Pascal-written software, prospective purchasers of Pascal processors, and many others, will also be interested in the validation suite. For them it will provide an opportunity to measure the quality of an implementation, and to bring pressure on implementors to provide a correct implementation of

Standard Pascal. In turn, this will improve the portability of Pascal programs. To emphasize this role, the validation suite also contains some programs which explore features which are permitted to be implementation defined, and some tests which seek to make quality judgements on the processor. The validation suite is therefore an important weapon for users to influence suppliers.

Naturally, implementors of Pascal are best placed to understand why a processor fails a particular test, and how to remedy the fault. However, the users' view of a Pascal processor is mainly at the Pascal language level, and the fact of a failure is sufficient for the users' purpose.

2. The test program structure

Each test program follows a consistent structure to aid users of the suite in handling them.

(i) Each program starts with a header comment, whose structure is given later.

(ii) The header comment is always immediately followed by a blank line and then an explanatory comment in plain English, which describes the test to be carried out and its probable results. The initial part of this comment starts with "(:" and is printed in the test report if this test does not produce the expected result.

(iii) Each program closes with the characters "end." in the first four character positions of a line. This pattern does not otherwise occur in the program text.

(iv) All program lines are limited to 72 character positions. Direct textual replacement of any lexical token, or the comment markers with the approved equivalents given in the Standard, will not cause the significant text on a program line to exceed 72 characters.

(v) The lexical tokens conform with the conventions set out in the draft ISO standard. Thus comments are enclosed in curly brackets, the not-equal token is "<>", etc. In addition, all program text is in lower case letters, with mixed-case used in comments in accordance with normal English usage. String-constants and character-constants are always given in upper-case letters. (Note: A few tests set out to check lexical handling, and may violate these rules. Translation of mixed cases to one case will therefore make these tests irrelevant, but will have no other effect.)

(vi) If a test has been modified in any way since the previous version of the validation suite, or is newly introduced in the current version, the explanatory comment for the test is followed by a version change comment, which commences "(Vv.x: ", followed by an indication of the change involved. "v" is the version number and "x" the number within that version, e.g. "(V3.1: New test.)". Next always comes one blank line, preceding the program heading. (Note: The version comment does not appear if the test is exactly as it was in the previous version.)

(vii) The program writes to the default file output, which is therefore

declared in the program heading. (Note: some tests do no printing including the minimal program).

Some programs need to break these conventions in order to test the desired feature. All these irregular tests are in one file. The reason for their inclusion in this file is recorded in the comment with the test. Processing this file needs care.

2.1 The header comment

The header comment always begins with the characters "(TEST" in positions 1-5 of a line (or "(PRETEST" in positions 1-8). No other comments are permitted to have the character "(" and "T" (or "(" and "P") directly juxtaposed in this way. The syntax of a header comment in Extended Backus Naur Form [7] is given by:

```
header-comment = "(" [ "PRE" ] "TEST" program-number ", "
                      "CLASS=" tail ")" .

tail    =    "LEVEL1, SUBCLASS="  category-name |
             "EXTENSION, SUBCLASS=" category-name |
             category-name .

program-number = number ("." number) "-" number .


number         = digit {digit} .


category-name = "CONFORMANCE" | "DEVIANCE" |
                "IMPLEMENTATIONDEFINED" reference-number |
                "IMPLEMENTATIONDEPENDENT" reference-number |
                "ERRORHANDLING" reference-number |
                "QUALITY" .


reference-number = ", NUMBER= " number .
```

For example, possible header comments are:

```
(TEST 6.5.3-10, CLASS=CONFORMANCE)
(TEST 6.6.3.7-1, CLASS=LEVEL1, SUBCLASS=DEVIANCE)
(PRETEST 6.5.4.2-1, CLASS=ERRORHANDLING, NUMBER=30)
```

The program number identifies a section in the Standard which gives rise to the test, and a serial number following the dash to uniquely identify each test within that section. If other sections of the Standard are relevant, the explanatory comment will mention them. The program identifier is constructed from the program number by replacing "TEST" by "t", "." by "p" for point, and "-" by "d" for dash. The replacement for "PRETEST" is "p". Thus the first header comment above belongs to a program t6p5p3d10 and the last one to p6p5p4p2d1. This technique may also be used to make the program source test file in processing.

The category-name identifies a class into which this test falls. The function and design of each test depend on its class. These are explained later. Thus it is possible to read through the validation suite file and simply identify the header comment by the leading "{T" (or "{P" for pretests) in the first two character positions, identify its section relevance and construct a unique identifier for each program, and to select programs of particular classes.

The reference number corresponds to a particular situation referenced in the Standard. These situations are given a numerical reference for convenience; see the relevant subsections below. There could be several or no tests corresponding to one such situation.

2.2 The program classes

An example of a test in each class appears in Appendix B with a commentary against each one.

2.2.1 Class = CONFORMANCE

The simplest category to explain is CLASS=CONFORMANCE. These programs are *always* correct Standard Pascal, and should compile and execute. With one exception (the minimal program), the program should print "PASS" and the program number if the program behaves as expected. In some cases an erroneous interpretation causes the program to print "FAIL"; in other cases it may fail before doing this (in execution, loading or compilation). Conformance tests are derived directly from the requirements of the Standard, and attempt to ensure that processors do indeed provide the features that the Standard says are part of Pascal, and that they behave as defined. Since conforming programs execute to completion, typical conformance tests will include a number of related features; all will be exercised by processors that pass.

2.2.2 Class = DEVIANCE

The next simplest category to explain is CLASS=DEVIANCE. These programs are *never* standard Pascal, but differ from it in some subtle way. They serve to detect processors that meet one or more of the following criteria:

(a) the processor handles an extension of Pascal,

(b) the processor fails to check or limit some Pascal feature appropriately, or

(c) the processor incorporates some common error.

Ideally, a processor should report clearly on all deviance tests that they are extensions, or programming errors. This report should be at compile-time if possible, or in some cases in execution. A processor does not conform to the Standard if it executes to completion. In such cases the program will print a message beginning "DEVIATES", and users of the tests must distinguish between extensions and errors. (In a few cases a possible extension is tested also for consistency under this class.)

It is obviously not possible to test all possible errors or extensions. The deviance tests are therefore generated from some assumptions about implementation (which may differ from test to test), and from experience with previous incorrect compilers. No attempt is made to detect extensions based on new statement types or procedures, but attention is concentrated on more stable areas. Obviously since each deviance test is oriented to one feature, they tend to be shorter than conformance tests, but also for each feature there is a short series of tests where one conformance test will contain several subtests.

2.2.3. *Class = IMPLEMENTATIONDEFINED*

In some sections of the Standard, implementors are permitted to exercise some freedom in implementing a feature. One class of such features which must be defined for any particular processor is called IMPLEMENTATIONDEFINED. The tests in this class are designed to report on the handling of such features. A processor may fail these tests by not handling them correctly, but generally should execute and print some message detailing the implemented feature. The collection of such implementation features is useful to the writers of portable software. The implementation defined features of a processor must be specified in the compliance statement that forms part of the documentation that is needed for compliance with the Standard. For convenience, each feature is numbered and the number is included in the heading comment of each test.

The reference numbering used follows the style of Appendix D of the Standard and is as follows (references are to sections of the Standard):

Numbering of Implementation-Defined situations

E.0 A complying processor is required to provide a definition of all the implementation-defined features of the language. To facilitate the production of this definition, all the implementation-defined aspects in clause 6 are listed here. See section 5.1 (d).

E.1 The value of each char-type corresponding to each allowed string-character. See section 6.1.7.

E.2 The subset of the real numbers denoted as specified by signed-real. See section 6.4.2.2 (b).

E.3 The values of char-type. See section 6.4.2.2 (d).

E.4 The ordinal numbers of each value of char-type. See section 6.4.2.2 (d).

E.5 The point at which the file operations rewrite, put, reset and get are performed. See section 6.6.5.2.

E.6 The value of maxint. See section 6.7.2.2.

E.7 The accuracy of the approximation of the real operations and functions to the mathematical result. See section 6.7.2.2.

E.8 The default value of TotalWidth for integer-type. See section 6.9.3.1.

E.9 The default value of TotalWidth for real-type. See section 6.9.3.1.

E.10 The default value of TotalWidth for Boolean-type. See section 6.9.3.1.

E.11 The value of ExpDigits. See section 6.9.3.4.1.

E.12 The value of the exponent character ('e' or 'E'). See section 6.9.3.4.1.

E.13 The case of each character of 'True' and 'False' for output. See section 6.9.3.5.

E.14 The effect of the procedure page. See section 6.9.5.

E.15 The binding of a file-type program parameter. See section 6.10.

E.16 The alternative representations to the reference representation. See section 6.1.9. This is not formally described as implementation-defined, but is clearly a defined property of an implementation.

2.2.4. Class = IMPLEMENTATIONDEPENDENT

The second class of features that can vary between processors does not have to be defined for any particular processor. Moreover, any program which depends upon the particular interpretation of such a feature is not a conforming program. Most of the features in this class relate to the order of evaluation of expressions within a single statement. The tests in this class determine the action taken by a processor in a single context. Hence these programs necessarily do not conform to the Standard (and so a processor can take any action). As with the implementation defined tests, these are numbered for convenience.

The reference numbering used follows the style of Appendix D of the Standard and is as follows (references are to sections of the Standard):

Numbering for Implementation-Dependent situations

F.0 A complying processor is required to provide documentation concerning the implementation-dependent features of the language. To facilitate the production of such documentation, all the implementation-dependent aspects specified in clause 6 are listed here. See section 5.1 (i and f).

F.1 The order of evaluation of the index-expressions of an indexed variable. See section 6.5.3.2.

F.2 The order of evaluation of expressions of a member-designator. See section 6.7.1.

F.3 The order of evaluation of the member-designators of a set constructor. See section 6.7.1.

F.4 The order of evaluation of the operands of a dyadic operator. See section 6.7.2.1.

F.5 The order of evaluation, accessing and binding of the actual parameters of a function-designator. See section 6.7.3.

F.6 The order of accessing the variable and evaluating the expression of an assignment statement. See section 6.8.2.2.

F.7 The order of evaluation, accessing and binding of the actual-parameters of a procedure-statement. See section 6.8.2.3.

F.8 The effect of inspecting a textfile to which the page procedure was applied during generation. See section 6.9.5.

F.9 The binding of the variables denoted by the program parameters to entities external to the program. See section 6.10.

F.10 The number and order of evaluations of the file parameters of the procedures read and write. Not formally defined as implementation-dependent, see section 6.6.5.2.

F.11 The number and order of the evaluations of the array parameters of the procedures pack and unpack. Not formally defined as implementation-dependent, see section 6.6.5.4.

2.2.5. Class = ERRORHANDLING

The Standard specifies a number of situations by stating that "an error occurs if" the situation occurs. The tests of this class each evoke one (and only one) such error. They are therefore not in Standard Pascal with respect to this feature, but otherwise conform.

A correct processor will detect each error, most probably as it occurs during execution but possibly at an earlier time, and would give some explicit indication of the error to the user. Processors that fail to detect the error will exhibit some undefined behaviour: the tests enable these cases to be identified, and allows for documentation of the handling of detected errors.

Each test is numbered following the references used in Appendix D of the Standard.

Corresponding to each test program in this class, there is a corresponding pretest. The pretest is placed immediately before each test. The pretest consists of a correct Pascal program written to be as similar as possible to the corresponding test. The intention of the pretest is to reduce the risk of the test program being (correctly) rejected by a processor but for the wrong reason. Success in the error handling test is judged by complete execution of both the test and pretest (if the error is not detected), or complete execution of the pretest but failure of the test (if the error is detected).

2.2.6. Class = QUALITY

These tests are a miscellany of programs which have as their only common feature that they explore in some sense the quality of an implementation, for example:

* tests that can be timed, or used to estimate the performance.

* tests that examine in detail the approximation used for values of the

type real and for operations on values of type real.

* tests that establish whether the implementation has a limit which is a virtual infinity in some list or recursive production. For example a deep nesting of for-loops (but not unreasonable!) would see whether there was any limit, perhaps due to a shortage of registers on a computer.

All quality tests conform to the Standard as far as possible. However, the real tests could be rejected by a processor without being incorrect since the algorithms depend upon details of the floating point system not supported by the Standard (see Chapter 5).

2.2.7. Class = EXTENSION

These are specific to some conventionalized extension approved by the Pascal Users Group, such as the provision of an otherwise clause in case statements. The subclass gives the purpose of the test according to the previously explained classes.

2.2.8 Class = LEVEL1

These tests are for level 1 of ISO Pascal; that is for conformant arrays. The tests should be run on a level 0 processor to check that they are rejected. The subclass indicates the purpose of the test according to the previously explained classes.

3. Structure of the validation suite

The validation suite as distributed consists of:

A. Machine-readable files

1. A header file containing the character set and an explanation of the structure of the other files.

2. A skeleton program, written in Pascal, to operate on the series of files in A.7.

3. A copy of this report.

4. The program Checktext (see Appendix B).

5. The program Assumptions (see Appendix C).

6. An index to the suite.

7. A series of files consisting of the sequence of test programs arranged in lexicographic order of their program-number within each class (see section 2.1)

B. Printed materials

1. This document.

2. A printed version of A.1.

The skeleton program as supplied prints the test programs on the output file, but calls a procedure **newprogram** before listing the start of a program, and calls a procedure **endprogram** after printing the last end of a program. These procedures as now supplied simply print a heading and a separator respectively. However, users of the suite may write versions of **newprogram** and **endprogram** that may write programs to different named files, and which may initiate jobs in the operating system queues to carry out the tests. The two procedures **newsuite** and **endsuite** are also provided in case these are of use.

Since **newprogram** may return a status result, it may also be programmed to be selective in its handling of tests. Only conformance tests may be selected, or only tests in section 6.3, as required.

The skeleton program is in Standard Pascal, and conforms to the conventions of the validation suite (but has no header comment).

4. Reporting the results

The results of passing the validation suite through a Pascal processor should be reported in a standard way, illustrated by the schema below.

```
PASCAL PROCESSOR IDENTIFICATION
      MACHINE:
      COMPILER:
      LEVEL:
-------------------------------
TEST CONDITIONS
      DATE:
      TESTS BY:
      TEST VERSION:
      REPORTED BY:
---------------
CONFORMANCE TESTS
-----------------
Number of tests passed = ?
Number of tests failed = ?
      details of failed tests:
      TEST ???? : explanation of why or what
          .....

DEVIANCE TESTS
--------------
Number of deviations correctly detected = ?
Number of tests showing true extensions = ?
Number of tests not detecting erroneous deviations = ?
      Details of extensions:
          .....
      Details of deviations:
          .....
```

```
ERROR-HANDLING TESTS
--------------
Number of errors correctly detected = ?
Number of errors not detected = ?
      Details of errors not detected:
          .....

IMPLEMENTATION-DEFINED TESTS
---------------------------
Number of tests run = ?
Number of tests incorrectly handled = ?
      Details of implementation-defined features:
          .....


IMPLEMENTATION-DEPENDENT TESTS
-----------------------------
Number of tests run = ?
Number of tests incorrectly handled = ?
      Details of implementation-dependent actions:
          .....

QUALITY TESTS
-------------
Number of tests run = ?
Number of tests incorrectly handled = ?
      Results of tests:
          .....

EXTENSION TESTS
---------------
Extensions present = ?
          .....

LEVEL 1 TESTS
-------------
Number of tests run = ?
  For level 0 processor, were all the tests rejected?
  For level 1 processor, were the expected results obtained?
```

5. Acknowledgements

The authors gratefully acknowledge the assistance of many colleagues who have collected difficult cases and bugs in their compilers and have passed them on for inspirational purposes. Many of the tests have been derived from work of B.A. Wichmann [3] and A.H.J. Sale [8] and significant contributions have also been made by A.M. Addyman, J. Miner, R.D. Tennent, N. Saville, C England, Z J Ciechanowicz and R. Freak contributed greatly by bringing consistency and care into the large effort required to assemble the validation suite itself.

The present level of the suite would not have been possible without the work of BSI DPS/13/4 in drawing up the ISO Standard, nor without the support of the Pascal Users Group.

6. References

[1] BS6192, Specification of the computer programming language Pascal, British Standards Institution 1982. (also ISO 7185).

[2] AFSC, User's Manual, COBOL Compiler Validation System, Hancom Field, Mass., 2970. (1970)

[3] Wichmann, B.A. & Jones, B. Testing ALGOL 60 Compilers, Software – Practice and Experience, Vol 6 pp261–270. 1976.

[4] Wichmann, B.A. Some Validation tests for an Algol compiler, NPL Report NAC 33, March 1973.

[5] DeMorgan, R.M., Hill, I.D., Wichmann, B.A. Modified Report on the Algorithmic Language ALGOL 60, Comp. J. Vol 19 No 4. pp364–379. 1977

[6] Wirth, N and Jensen K, Pascal User Manual and Report, Springer-Verlag 2nd edition, 1975.

[7] BS6154, Method of defining syntactic metalanguage. British Standards Institution (1981)

[8] Sale, A.H.J. Pascal Compatibility Report (revision 2), Department of Information Science Report r78–3, University of Tasmania, May 1978.

5

Developing the testing procedures

B. A. Wichmann and Z. J. Ciechanowicz

1. Checking the testing

The general nature of the test suite has been described in the preceding papers. However, running the test suite is a significant undertaking which can be substantially eased by other aids. Firstly, a program has been developed which checks for basic assumptions upon which the test suite relies. For instance, it is assumed that the range for integers is at least -30000..30000. Running such a program can avoid an abortive attempt to run the entire suite. This is listed in Appendix C.

A second example of a testing aid is more vital. The test suite must be mounted onto the computer being tested. As yet, loading this material via a data communications line is not viable (and in any case, there is no requirement to have a communications facility with every Pascal compiler). On the other hand, media conversion is a well-known trouble spot for effective portability of software. Hence the possibility of incorrect conversion cannot be ignored, especially since the conversion of the suite to one case of characters and substitution of { by (* is permitted. The solution to be adopted to this is a program which constructs a 96-bit *parity* check on files or programs in the suite. This parity check is independent of the permitted substitutions. Comparison between the parity of the delivered software and the converted text ensures that everything is correct. This parity checking program is written in Pascal (of course). For the technical details see Appendix B and [1].

2. Simplification of testing

With over 400 individual programs in the test suite, methods which can ease the burden of testing must be welcome. Of course, each test is written according to fairly rigid standards and this itself aids testing substantially. For instance, particular subsets of tests can easily be retrieved for execution. Suppliers of software are constantly faced with the problem of revalidating a new release of an item of software. The standard practice is to have a special job which re-runs all the previous tests and checks for differences in the results. Unfortunately, such techniques depend upon the underlying operating system and are hence inappropriate to the context of this project. All that we can assume is a Pascal compiler.

Although the test suite itself is quite large (about 300 pages), the output is modest. Most tests produce only a single line and at the most it is much less than the source listing of the program. Moreover, from the point of view of validation, it is rare to require more than one line of output. Naturally, programs which should not compile must produce *no* output on execution. It is important to note that inspection of the compiler listings is not needed.

An automatic testing facility has now been produced which will be available with the next version of the suite (probably only from BSI). The facility consists of a number of fairly small programs which firstly produce a standardized description of the test suite and the output from a test run, and then produce a standard report. The facility uses initial comments in the test suite itself to determine the nature of messages produced in the report when a failure is detected. In this way, the report is entirely automatic and hence devoid of subjective judgement. In consequence, it produces an indication of a test program that failed without any indication of why it failed. Hand produced reports invariably give the reason for the failure but it is often impossible to independently substantiate this. The facility makes the assumption that all the output from the execution of the programs in the test suite can be collected in one file.

The main purpose of the automatic testing is to simplify the routine procedure for validation that BSI will undertake. This should reduce the cost of validation to a level where re-testing every year or two becomes an attractive option. Retesting of COBOL and FORTRAN in the United States is undertaken because of Federal procurement policy whereas we must ensure that retesting is economic in a competitive market for Pascal compilers.

2.1 The automatic analysis of the tests

In order for automatic analysis to be feasible, some assumptions must be made about the analysis procedures. In particular:

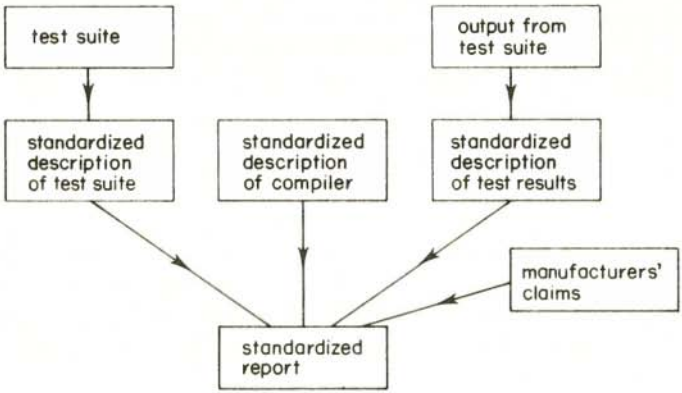
1/ All the output from a validation must be collected in one file

2/ Inspection of compilation listings is not needed

3/ The reason for failure of a particular test cannot be given.

Thus, an automatically produced report will contain such items as: a 'description' of the compiler; manufacturers' claims regarding their compiler; for each failed test an indication of the particular feature being tested; some sort of statistical analysis.

The overall structure of the automatic analysis package is given:



The first section of the report contains a highly stylized description of the compiler, the details of which will be supplied by the manufacturer when the compiler is submitted for validation (e.g. which error conditions will be detected). This will be followed by a list of manufacturers' claims regarding such things as extensions etc. The last section of the report contains details of all the failed tests i.e. test number together with a description of the test. Error handling tests will be treated differently since a manufacturer has the option of not detecting specific error conditions. The manufacturer must indicate which errors the processor will detect and this is checked by the automatic analysis procedures.

An outline report will look as follows:

```
STANDARDIZED COMPILER DESCRIPTION
   e.g. level 0 or level 1
        ability to detect errors
      .
      .
MANUFACTURERS ADDITIONAL DESCRIPTION
   e.g. extensions
        limitations
      .
      .
      .
DETAILS OF TESTS


CONFORMANCE TESTS

Total number of conformance tests =
```

```
Number of tests passed =
Number of tests failed =

Details of tests that failed to conform to Standard:
   .
   .
   1.1-1        This test examines if empty records
                are allowed by the processor.
   .
   .
   .
DEVIANCE TESTS

Total number of deviance tests =
Number of tests which detected deviations =
Number of tests which did not detect deviations =

Details of tests which deviated:
   .
   .
   .
ERROR HANDLING TESTS

Total number of error handling tests =
Number of tests which detected errors =
Number of tests which did not detect errors =

Details of tests in which errors should have been detected:
   (in these tests,according to the manufacturer, the processor
    should have detected the error but was unable to do so)
   .
   .
   .
Details of tests with other undetected errors:
   .
   .
   .
IMPLEMENTATION DEFINED TESTS

Total number of implementation defined tests =
Number of tests passed =
Number of tests failed =

Details of successful tests:
   .
   1.1-2        THE IMPLEMENTATION DEFINED VALUE OF
                MAXINT IS       2147483647
   .
   .
Details of failed tests:
   .
   .
   .
QUALITY TESTS
```

```
Total number of quality tests =
Number of tests passed =
Number of tests failed =


Details of failed tests:
    .
    .
    .
```

3. Specific test problems

The Pascal Standard is the result of an international agreement. The agreement is essential if the Standard is to be effective and yet reaching such an agreement means making compromises due to the conflicting needs and objectives of the parties concerned. Hence when testing to the Standard, one must expect some problems in view of the wide ranging nature of the tests. If only valid Pascal programs were in the suite, then these problems would be reduced, but so would be the usefulness of the suite.

Let us consider a representative sample of these problems:

3.1 Reserved words. The Standard is not quite clear as to the extent to which an implementation can add to the list of reserved words. The list in the Standard is clear enough but if carte blanche is given for additions, the programmer cannot write a single identifier in his program. This is a case where the Standard has attempted to acknowledge the existence of extensions but has done so in a way that has made the Standard less accurate.

3.2 Line length. The Standard gives no indication of the physical nature of source text lines. In particular, an implementation is apparently free to restrict the line length to, say, 10 characters, making portability very difficult and the "correct" implementation less than useful. Fortunately, in practical terms this does not appear to be an issue as no system is known to us which does not support 72 characters, even when the longer options for { and [are substituted. Some systems ignore characters after column 72 for line numbering. This fact is not detected by the suite.

3.3 Obscure issues. In some cases, the interpretation of the Standard is in doubt and in other cases, even when it is not, an extreme case could cause problems for any practical compiler. As an example of a difficulty in interpretation, consider a program ending in the characters:

```
end.)
```

On one interpretation, **"end."** completes the program and the extra ")" falls outside the program (and the Standard). On another interpretation, ".)" is equivalent to] and **"end]"** is not a correct ending to a program which therefore makes the program invalid.

As an example at a similar lexical level of an issue where the interpretation is not in doubt, consider the assignment of an integer J with 10 divided by 2:

```
J := 10 div 2;
```

The Standard requires a space between the "10" and "**div**" and yet almost no actual compiler requires this. Hence the invalid program without the space is not rejected. Does this matter? Of course not, but the Standard is specific on this point.

3.4 Conformant arrays. The Standard has two levels – with and without conformant arrays. This makes the test suite more awkward to construct since tests for conformant arrays must be carefully separated. In a validation, all tests are run, but compilers not handling conformant arrays must reject these additional tests. Hence this issue does not present any real difficulties.

3.5 "Quality" issues. Many tests in the suite address such issues as the size and complexity of programs that can be handled by an implementation. These issues are specifically excluded from the Standard, but affect every part of the suite and hence are discussed separately below.

4. Quality Issues

A Pascal implementation can be formally correct and almost useless as a practical tool. We have discussed one example of this: a hypothetical compiler which will accept only lines of less than (say) 10 characters. Another example is an implementation with `maxint` having a small value. Neither of these issues is of practical significance because all implementations have reasonable values for these parameters. More difficult is the question of the complexity of a program that an implementation will accept.

Every implementation will have some limits. The size of code generated naturally imposes some limit. A compiler may impose other limits. These restrictions should not have the effect of distorting the user's programming style. A questionnaire was sent to members of the ISO Pascal group asking for suggestions on what the numerical values of certain limits should be. For instance, *for*–loops are checked to a nested depth of 15, in one specific quality test. Naturally, the rest of the test suite is conservative in the use of nested *for*–loops. There are about 30 such complexity tests in version 3.0 of the test suite. It is important to note that a failure in a quality test does not mean that a compiler is incorrect. Hence the purpose of including such tests is to demonstrate that an implementation is a practical tool and not just a correct implementation. It has been possible to set the level of these tests quite high because most compilers are implemented in Pascal. This means that there cannot be any small limits on program size.

Another class of quality tests concerns real arithmetic. The Standard merely requires that an implementation provides the data type `real`, an approximation to real values. The nature of this approximation is not specified. Hence, in principle, apart from the usual floating point representation, a fixed point representation, or a purely logarithmic representation could be used [2]. Fortunately, all implementations appear to provide, at least as an option, a floating point representation. Now that IEEE has defined a standard floating point system [11], one hopes that good numeric facilities will be widely available on microprocessors.

To make any detailed use of floating point requires that one makes additional assumptions about the representation used. The test suite uses the algorithms of Cody and Waite [3] to determine the characteristics of the machine. These routines had to be changed slightly since the Multics Pascal system gave the wrong results, but the version in 3.0 is believed to be satisfactory. Cody and Waite list extensive test routines in FORTRAN for the standard functions which have been transcribed into Pascal for version 3.0. Their work also indicates the accuracy that should be expected, and these criteria have been added to the Pascal versions to give pass/fail messages. It was necessary to revise the random number generator used in version 2.2 in order to allow correct execution on 16-bit implementations [4].

Tests have been included for real input/output. Unfortunately, Cody and Waite did not consider these algorithms and no machine independent criteria are known for the expected accuracy. This issue is currently being investigated by the numerical analysts at NPL.

The basic real operations are checked in a very simple way by a few tests in the suite. These tests exploit the fact that small integers are handled without error by all known floating point systems. A much better test facility is available from Bell Laboratories [5], but this is too big to be added in its current form to the test suite.

The last aspect of quality addressed by the test suite is that of "performance". Since there is no actual timing performed, these tests are merely the basis for performance measurement rather than genuine performance tests. The majority of the test suite programs are very unusual in their use of machine resources and in the nature of the Pascal test program itself. The performance tests are more representative, at least in the area that is being measured. There are only four such programs at present, and hence further ones are solicited. The existing ones are a synthetic benchmark [6], a mathematical array access test [7], a procedure call test [8] and a binary matrix test.

5. Enhancements to the suite

Version 3.0 of the test suite is incomplete in several respects. However, the overall approach is thought to be satisfactory, which is confirmed by the fact that the suites devised by others for Ada [9] and CHILL [10] are derived from the Pascal suite.

Several enhancements are necessary. For instance, the tests for conformant arrays are to a previous draft of the Standard and need revising for the new Standard. This is the only area known to require significant revision. (The test suite was issued in this form because relatively few implementations handle conformant arrays and hence a delay because of this was not worthwhile.)

The automatic analysis package requires small changes to most of the suite. For instance, the error-handling tests must indicate the error number being tested. Undetected errors can then be checked against the compiler specification as errors that an implementation will not necessarily detect.

Of course, testing can never show the absence of errors. However, one

can ensure that the test suite exercises all the code of a particular Pascal implementation. A preliminary trial on the Welsh checker [13] indicated that about 50 more test programs were needed, all concerned with error conditions. It is hoped to repeat this with an execution facility so that all aspects of the run-time system are exercised as well. Systematic construction of syntactically erroneous programs does not seem reasonable because the volume of such tests would be too large. In fact, errors in the syntax recognition of Pascal compilers are not so common and are concentrated in areas such as the differences between Jensen and Wirth and the Standard.

Another method of checking for completeness of the suite is to discover if there is a conformance and deviance test for each "shall" in the Standard. This will be done when the final version of the Standard is available in computer-readable format.

Not surprisingly, we have been given hundreds of suggestions for further tests. Indeed, many of the existing tests were as a result of input from interested users. Unfortunately, it takes some time to process these suggestions. We need to check that no such test exists already and that the test does indeed check the aspect stated. Usually, the tests need to be rewritten to conform to our conventions.

6. Conclusions

Although testing can never show the absence of bugs, the Pascal test suite has the overriding advantage of being simple to use on any new or modified implementation. If this project is successful in encouraging the validation of Pascal compilers, then the main advantage will be the increased portability of Pascal programs. On the basis of a study of bug reports from existing compilers, the test suite is likely to make compilers about twice as reliable (assuming that other forms of testing are still carried out).

It is interesting to contrast testing with the work of Polak on formal verification of a Pascal compiler [12]. The verification was not to ISO Pascal, but even if it was, to check this would be a non-trivial task. Moreover, a verification may well rely upon assumptions about the underlying hardware and operating system which would be virtually impossible to formally verify. In fact, tests such as those for Pascal do act as a check on the hardware (and operating system) which is useful in itself, independent of any question of validation of the software. For those who are not convinced that good software tests can reveal errors in hardware, the appendices in the Bell Laboratories report [5] should be consulted. Yet these errors are in the comparatively well-understood area of floating point.

References

[1] B A Wichmann, Has the program been altered? Software - Practice and Experience, Vol 11, No 8, August 1981 pp877-879.

[2] K A Simms, A new number language for scientific computers. Dr Dobb's Journal of computer Calisthenics and Orthodontia, Vol 5 no 10, 1980.

[3] W J Cody and W Waite. Software manual for the elementary functions. Prentice-Hall, 1980.

[4] I D Hill and B A Wichmann. An efficient and portable Pseudo-Random number generator. NPL report (to appear).

[5] N L Schryer. A test of computer's floating-point arithmetic unit. Computer Science Report No 89. Bell Laboratories, NJ 07974. 1980.

[6] H J Curnow and B A Wichmann. A synthetic benchmark. Computer Journal Vol 19, 1976 pp43-49.

[7] J du Croz and B A Wichmann. A program to calculate the GAMM measure. Computer Journal Vol 22, 1979 pp317-322.

[8] B A Wichmann. How to call procedures or second thoughts on Ackermann's function. Software - Practice and Experience. vol 7 1977 pp317-329.

[9] J B Goodenough. The Ada compiler validation capability. ACM Sigplan Notices. Vol 15, no 11, pp1-8 (1980).

[10] H Gutfeldt. CHILL Compiler validation. Hasler AG. Contribution to CCITT study group XI, 1981.

[11] D Stevenson. A proposed standard for floating point arithmetic. Draft 8.0 of IEEE Task P754. Computer Vol 14 pp51-62, 1981.

[12] W Polak. Compiler specification and verification. Lecture Notes in Computer Science. No124, Springer-Verlag 1981.

[13] J Welsh. A standard compiler for Pascal. Chapter 7 of this book.

6

Second thoughts on the validation suite

A. H. J. Sale

1. Introduction

The Pascal Validation Suite was developed simultaneously with the ISO Pascal Standard, and greatly influenced the development of the Standard, as well as being based on the Standard's prescriptions. It has been supplied to over 300 organizations, and has been widely used by implementors of Pascal to test out their compilers and Pascal systems. The background to its construction has been stated in many places, not least in a special issue of Pascal News.

The importance of validation should not need stressing, but is not always realized. A validation test for a software system is like a roadworthiness test for a car, or an airworthiness test for an aeroplane: it is unthinkable that an untested system would be loosed on the public except in very primitive and backward corners of the world. The lack of validation tests of value for most software systems is deplorable, and to it we owe the generally poor state of compilers and systems for most programming languages today, such as BASIC, FORTRAN and even COBOL. Of course validation tests do exist for some of these languages (the COBOL tests being of very long standing), but the standard for COBOL virtually emasculates its own validation, as I shall point out later.

2. Foundations

It should not need saying that the construction of a validation suite is a feat of *software engineering*, not of computer science. No amount of testing could *prove* a Pascal processor to be correct, but then no amount of testing can prove that a plane is airworthy (at least not without knowledge of the internal structure, which is equivalent to knowing a correctness proof for a compiler).

However it has been a major concern to ensure that in the construction of the PVS (as I shall refer to it) as much formal structure is embodied in it as is possible. I shall try to explain how this was done.

2.1 Test categories

It was realized very early in the design of the PVS that many of the existing validation procedures for other languages were tests for conformance with the requirements of the language as set down by the Standard: clearly Pascal would have similar requirements and so the category of conformance tests was required. However, the Pascal Standard also laid down that everything not allowed was expressly forbidden, and so it was obvious that the PVS would also have to include a category of test which explored this forbidden region.

Tests are those which are *not* in Standard Pascal, but which might be expected to slip through the defences of some Pascal systems, implying that the system failed to adequately check what the language required, or that the axioms were violated. Imagine the possibilities if someone were to claim the relaxation of the strong typing rules as an extension (it has happened!), or to allow the violation of the axioms of integer arithmetic (which has also been detected). Accordingly, after a lot of thought, this category of deviance tests was established; the name suggests that the processor is abnormal and deviates from socially acceptable behaviour ...

In an ideal world, the conformance and deviance tests would cover the whole set of possible programs and their input data, and nothing more would be required. However, Pascal, is not perfect, and it explicitly recognizes this fact in its definition in the Standard by making allowance for imperfections of processors. This gives rise to two additional categories of tests, required to explore the grey areas between what is thoroughly well-defined and what is absolutely banned.

The simplest category to understand is that of implementation-defined tests which were introduced to permit Pascal processors some freedom of implementation. There are not many of these, and some of them are regrettable but due to early laxity in the implementation of Pascal. Others are more obvious to the average programmer (such as the implementation defined value of the pre-defined constant `maxint` which is the biggest integer for which the integer arithmetic axioms hold). However, careful inspection of all the implementation-defined features yields the conclusion that they all represent lapses from abstraction, whether to accommodate differing implementation or implementor's laziness. It is fortunate that there are so few of them.

Since each implementation-defined feature must be explicitly named, it suffices to construct a test for each one; since programs containing the implementation-defined feature are still correct (or may be so provided they do not violate some other rule) the test may be able to explore the behaviour of the feature and report on it.

The last category of test covers those features of Pascal which are regarded as conditionally correct. The idea of conditional correctness in Pascal gives rise to two kinds of test: implementation-dependent, and error-handling. The category will be treated further later.

At a late stage, a further category of tests was added to the PVS though they properly do not belong in a validation suite: these were the quality tests, which attempt to assess the quality of the implementation in directions which are not laid down by the Standard but which may be directed by it. For example, how accurate is the real arithmetic? Or the arithmetic functions? Is there a small limit on the allowable nesting level of procedures? Or for-statements?

The basic situation is set out graphically below: the test categories of the PVS are intended to reflect both the nature of the language and the formal structure of the kind of programming language Pascal is.

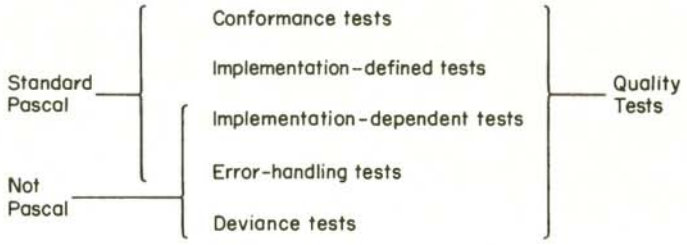


Fig. 1. Illustrating the relationship between categories

2.1 Conformance tests

Since all conformance tests are correct Pascal, it would be possible to construct a single very large program which contained most of the features; if it operated correctly then the processor passes the test, otherwise it fails. The only tests which could not be handled in this way are those which depend on the very structure of a single program, such as the minimal Pascal program:

```
program A;begin end.
```

Clearly this is undesirable from the point of view of isolating faults, since the probability that an untested processor will pass is vanishingly small. However it is the way most previous conformance validation tests have been constructed. In the PVS, however, the tests are broken down into cognate groups such that each conformance test is concerned with one small part of the definition of Pascal: it might test a number of facets of that part, but anything else it uses is simply an accident of construction. Such accidents of construction are necessary, of course, and sometimes it happens that a processor will fail a test by tripping over such an accident rather than the tested feature! This is desirable for a searching set of tests, but undesirable if the results are to be processed automatically.

The tests are also organized so that syntactic features are separated from semantic ones; thus the testing that all forms of logical expression are allowed is a separate test from one which tests the axioms of boolean algebra. In either case the problems of emptiness and infinity arise: the tests must include the limiting case of empty constructs (or one if that is the lower limit - Pascal has an ambivalent attitude towards zero and one), but clearly cannot contain the infinity case. To cater for this the "one, two, three, many" rule was designed, so that when a recursive or iterative definition did not have a limit, the test would contain a sample of the small constructs from one (or zero) up to three, and then a "many test" designed to have a repetition or recursion which was considered to be a lower bound for reasonable implementations.

This question of bounds for iterations is an important one, which has been neglected in the evolution of the Standard as a consequence of the muddling of language, implementation and program standards. One can distinguish several reasonable types of bounds:

(a) a lower bound which it is *expected* all implementations will be able to handle. This bound is used in the PVS to ensure that accidental constructions do not fail by reason of exceeding such a limit; of course tests organized to explore this point may quite reasonably exceed it.

(b) a lower bound which no reasonable implementation would go below. The conformance tests check that this is so, and an implementation that fails is regarded as being simply out of tolerance limits. For example a limit on nested with-statements of three, or a real number implementation with only two values (0 and 1) would be unacceptable. Judgement is needed to set reasonable bounds for each case.

(c) an upper bound beyond which it is irrelevant what the bound is. The term *virtual infinity* was used to describe this limit: any processor which exceeds it has a virtually infinite limit, and it is presumed that the incidence of programs which exceed the bound is vanishingly small. For example the Burroughs compiler limit of 29 lexically nested procedures has never been exceeded except by test programs.

The first bound is an assumption made for convenience in the construction of the tests; the second is regarded as a compulsory requirement which should be (but is not yet) written into a Standard for conforming Pascal processors. The third appears only in the quality tests, so that a processor which exceeds the bound is regarded for that purpose as being of high quality.

2.2 Deviance tests

In early versions of Standards for FORTRAN and COBOL, the Standards explicitly stated that they had no business with features or treatment not explicitly specified, limiting themselves therefore to defining *only* the conformance characteristics of the language. Obviously early validation tests for these languages were solely composed of what we have chosen to call conformance tests.

In more recent times, COBOL has moved in the direction of trying to control the area of language constructs outside the defined language. While still permitting freedom to extend the language in any way thought useful, the form of the extension is constrained by two requirements:

(a) the extension must not invalidate any standard construct or its meaning, and

(b) no extension shall be required in order to perform some standard action or make some standard construct.

Unfortunately, requirements have a very serious flaw: how does one go about constructing a test for them? It seems that it will be necessary to repeat validation tests for COBOL both will all switches enabling extensions turned OFF, and then again with all permutations of such switches enabling some or all extension features.

The Pascal Standard however tries to explicitly ban all deviations, and requires that extensions be regarded as deviations (with a hint of undesirability) by Pascal processors.

Constructing deviance tests is a quite different job from constructing conformance tests. While conformance tests call for a logical mind in exploring possibilities, and some deviousness in constructing allowed but rare usages, deviance tests deal with an infinity of possible constructions: or at least the infinity of deviant programs feels as though it is a *larger* infinity (in the Cantor sense) than that of all conforming programs. A high degree of deviousness and querying is called for to construct the tests.

The following basic techniques were used in the construction of deviance tests:

(a) Where the Standard specified a limit (say a minimum of one element in a construct), a test to violate the limit was introduced. A good example is the compulsory digits surrounding the decimal marker in a real constant.

(b) Where the Standard specified an axiom or requirement, explore the consequences of violating the axiom and construct tests which rely on possible violations of the axioms.

(c) With knowledge of parsing techniques, consider likely transformational errors arising from simplification of grammar, or simply oversights, and construct tests for these.

(d) With knowledge of machine architectures in general, and code generation techniques, consider likely misgeneration of code and possible oversights. Good examples are in the implementation of the for-statement, and in the area of boolean operators. The first is very complex and therefore liable to error; in the second many implementors are led astray by the so-called "logical and" and "logical or" instructions which are really nothing of the sort being implementations of set operations on cells.

(e) With knowledge of the parentage of many Pascal compilers, develop tests for the known bugs in the ancestor compiler. Good examples derived from Pascal-P are the incorrect scope treatment, the flawed identifier-length implementation, and various uses of (internal) undefined values.

(f) From bug reports on various compilers, select tests which look as though they may detect errors on a reasonable subset of compilers, or which are horrendous in their implications.

It will be clear that there will be a lot of deviance tests; since each test must contain *only one* deviation this also leads to proliferation. The techniques (a) to (e) are reasonably respectable and easy to implement; however a serious problem arises with category (f). Every detector of a bug thinks that his discovery is worth including in the PVS; rarely is this so. We have taken the attitude that idiosyncratic errors which are due to one compiler will not be included unless they do one of the following things:

* they reveal an overlooked area falling into the (a) to (e) techniques, or

* they are reasonably likely to be searching tests for a reasonable sized sample of Pascal processors, or

* the error thus shown up is so terrible that it ought to be held up for all to see and wonder at (and not to fall in the same trap).

Which focusses on an important point. The PVS is not only for an implementor to use to check or certify an implementation, nor only to include users in this checking and certification of systems. It serves an important role in publicising errors, and eradicating them, much as magazines such as **Choice** do in the consumer field. In this role it is serving to re-orient implementors' thoughts: extensions are generally bad; obedience to axioms is important; switches to turn off non-standard behaviour are essential; and so on.

However, the importance of the need to resist the encroachment of numerous ad-hoc tests into a validation suite cannot be over-emphasized. It is emerging as one of the major problems in the further evolution of the PVS; a single maintainer can maintain the necessary strength of will and purpose but it is much less likely that a committee (such as the Pascal Working Group ISO TC97/SC5/WG4) will be able to be strong enough. To illustrate the sort of test that one might be considering, consider the following program fragment:

```
var
   a,b,c:integer;
...
begin
   ...
   c:=round(sqrt(a*a + b*b));
   if c*c = (a*a + b*b) then
      {commented out code for time purposes};
   ...
```

One compiler noticed that the value of c was assigned in one statement, then immediately needed in the next boolean expression, so it saved a LOAD by leaving the value of c on the top of stack after assignment. Unfortunately, it then analysed the if-statement and found that (a) the boolean expression had no side-effects, and (b) the then and else clauses were empty, and it did not generate any code for the if-statement. The consequence was that the stack size thus grew one cell for every iteration of this code, until the program crashed for lack of memory.

The probability that this error would occur in another compiler is not vanishingly small, and yet it must be admitted that it is a somewhat irregular thing for which to test.

2.3 Conditional tests

By far the nastiest area of the Pascal Standard is the area of conditional correctness of programs. The Standard is confused over whether it is defining a language, a conforming processor, or a conforming program (though certainly less so than any other international Standard), and in this area it comes down firmly in favour of processors and their implementation.

There are two concessions to processors:

* implementation-dependency
Although this term is confusingly similar to *implementation-defined*, it is completely different. The Standard allows features so described to be implemented any way at all, including no interpretation or an erratic interpretation. Thus implementors of Pascal are given freedom of order of evaluation of expressions provided the effect is preserved (which includes parallel evaluation, or pre-evaluations), and freedom of interpretation of some effects.

The implications for users are severe: no reliance may be placed on the feature at all. However, since this term is applied to semantic features of the language, the user is generally placed in the situation of having no check by the processor that the feature is used within its bounds of correctness (i.e. without reliance on particular interpretations). The Standard and the PVS try to remedy this very unsatisfactory state of affairs by insisting on a statement by implementors in their documentation, and by testing the features so described.

* error-handling
This appallingly badly named facility in the Standard allows certain features to be termed *errors*. Such features are characterised by having at least one requirement for detection that cannot be determined unless the program is inspected together with its input data and its environment (or in loose terms which will suffice, if it is executed). Thus the detection of an error may occur during processing of a program by a compiler, but there are always some cases which cannot be detected without execution-time action.

Both features I term forms of conditional correctness because it is quite clear what a processor must do to conform to the Standard, but it is not at all simple to determine whether a program conforms. For implementation-dependency, the analysis must include the uses of all features so labelled (which involves the analysis of all expressions for side-effects); for errors the analysis cannot be completed unless the data and environment are known and a program may indeed be "correct" for a subset of all possible values of these.

Both features are lapses of abstraction, and were introduced in order to make concessions to implementors and machines. It is a tribute to Pascal that there are so few tests of these kinds, and one would hope that any future languages would have even fewer. The PVS tackles each example by a test which is correct Pascal except for a single exercising of the error or implementation-dependency. Very recently, each such test has been preceded by a "pre-test" which is closely similar except that the error or implementation-dependency does not occur. The purpose of this is to ensure that accidental failures are not diagnosed as successful error-handling: such is now diagnosed only if the pre-test works, but the error-test does not.

3. Quality testing

The size of the PVS, both in terms of number of programs and in terms of lines of code, is now very large. The original inclusion of "quality-oriented" programs as an intrinsic part of the suite is now seen as a mistake. These

tests are not derived from the Pascal Standard, and do not lend themselves to GO/NOGO treatment, and it is my conclusion that they should be regarded as a separate entity in the testing facilities of BSI. Examples of quality are

* those exploring the properties of real arithmetic

* those exploring the arithmetic function accuracy

* those exploring the finiteness of iterative or recursive definitions

* those exploring the quality of diagnosis of syntax or semantic flaws in programs

* those giving opportunity for comparing resource utilization (space, speed, etc.) in various phases of processing.

It should be possible for a Standards Committee to set minimal requirement in most of these areas, and perhaps even a "Gold Seal of Approval" level of requirement, so that processors which fully meet the mandatory requirements of the Standard could seek an additional mark of quality assurance.

Little emphasis has been placed on quality programs in Tasmania, as this area has been left for NPL and as enforcement of the Standard was seen as a higher priority.

4. Management

The PVS project grew out of individual and frankly scrappy collections of tests by B A Wichmann and A H J Sale. In the initial collections, the Wichmann tests were largely conformance tests and the Sale tests largely implementation-dependencies and errors. The merging of the two sets called for a rethink of the aims (as has been outlined) and a strategy for managing the tests.

This strategy has evolved slightly in the intervening years, but is still substantially the same. It consists of a test numbering scheme based on the relevant section (or most relevant section) of the Standard and a serial number within that (e.g. 6.4.3-2) and a number of program requirements. Each program has a standard header consisting of:

* a stylised comment incorporating the test number, its category, and other information

* a stylised comment incorporating the last revision notice

* a program header incorporating the number as part of the program name

* a plaintext comment explaining the purpose of the test.

Each program ends with the characters "end." on a line by themselves.

Within the program there normally appear write statements. These carry the test number, and information regarding the test performance. In the case

of conformance tests, successful completion results in the message bearing the word PASS. If the word does not appear, or in some cases if failure can be diagnosed without the program crashing when FAIL is printed, the test is failed.

In the case of a deviance test, the program should never even get to execution, so the production of any output, or the specific DEVIATES message, is a sign of failing the test. A different message was deliberately chosen.

The cases of the other test categories are similarly treated; however, implementation defined features require also a report on the feature's treatment and extra text is likely to be printed.

Clearly this organization calls for internal consistency in the test itself, and for no duplications of tests in the suite. To aid in this process, and since tests are often re-classified as the standard changes, a set of checking programs was written to report on inconsistencies, and these have been successful in maintaining a high degree of correctness in around 400 programs. Remember that the correctness of the PVS must be of an order higher than of the processors it tests, and preferably perfect.

With hindsight, it can be observed that although the decision was probably correct at the time, the size of the PVS now demands a better approach to management. I now consider that we should have defined a "test language" which incorporated the essential elements, and written a test generator which would have created tests from the test-language with guaranteed internal consistency. A simple implementation would have been no more than a macro-generator, but more complex organizations would be possible. If this had been done then the job of modification of tests would have been made *much* easier; a fact NPL I am sure will verify.

If the project were to be started again from scratch, I would be very tempted to construct it as a micro-based system, with a built-in data-base and test language facility, with the advantage of being able to transport the tests readily to a variety of test environments. Statistics on the frequency of failure of particular tests (and a fingerprint of test failures) would be very interesting.

5. Importance and demise

I cannot emphasize enough how important I believe the establishment of a test facility for software is. The PVS and the Pascal Standard *together* offer an important opportunity to clean up the mess that users continually see in vendor's software and particularly in compilers. If they can improve portability, and the abstraction level of Pascal is certainly better than anything else we have seen hitherto, then users will benefit considerably. Unfortunately, it would have been better had we had the Standard and the PVS operational four years ago... Before there were so many implementations.

However, I should like to add a note of warning: this style of testing has serious limitations, and it is ironic that one of our research projects in Tasmania is creating a Pascal processor which the PVS cannot test. Since I believe that the directions taken by this work are very important and likely to influence the whole course of language development, I have to say that the eventual demise of validation suites as we know them is foreshadowed.

However, much as I would like to say that Ada is probably one of the last big mistakes, I think the working out of the change will probably take us up to the year 2000 AD.

The trend to which I refer is brought about by the splitting of the software field in modern times into two major fields: that of the primitive application generators where interacting users define tasks for themselves, and that of provably correct software making up the substructure that supports the user-oriented levels. The application generators at present show a very naive face, but will no doubt improve in correctness checks and abstraction with time, but I am concerned with the second area.

For too long we have regarded programs as objects of *text*, to be manipulated as though they were English (or American or French or Japanese) prose; the assumption is built deeply into the design of Pascal, for example, and even more disastrously in Ada. But the programs are structured objects, not prose, and intelligence in editors can enforce such structure. When this is done, the markers and words that show off such structure can be seen as trivia, serving only to bind the *deep structure* of the program in a text form (where a text form is required).

In the Tasmania Intelligent Editor, the structure of Pascal is known to the editor in terms of syntax tables; users are prompted by the editor to exercise the available choices, and to elaborate programs; what is stored is simply the user choices. It is pointless to store keywords, delimiters, etc., because these are already known from the syntax: all that is needed is a trace of the traversal of the syntax which constructs the program tree, and a knowledge of any introduced leaves. Such programs are thus never stored as text, but may be displayed on the screen as such on demand, or even so generated for printing. Compilers need no error-recovery (or little) since the program is guaranteed to be structurally correct (if possibly incomplete). But for our purposes note most significantly that there is no way to feed the *text* of the Pascal Validation Suite into the system to test it, yet it is undeniably a Pascal processor. Thus the PVS already fails to test some Pascal processors.

Of course, in a full implementation of such a Pascal processing environment, besides the *exporter* program which converts a structured-form program to linearized flattened text for export to other systems or humans, there will have to be an *importer* program which accepted linearized flattened text programs from elsewhere and attempts to construct structured forms of them where possible. Would this give a key for the PVS? No, or not entirely, for much of what the PVS would be testing would be the correctness of the importer program which is surely laudable but not the same as testing the whole processor. Depending on how many semantic constraints were built into the syntax, there might be rather little to be done or tested in the compilers or interpreters that were common to imported and internally generated programs.

Worse still for the future of validation, once deep structure is fully realized then dialecticism will grow, and can be discouraged, without harm to portability. There are now French dialects of Pascal; there should be Malaysian, Japanese, English and Italian dialects as well as the existing Euro-American version. There could even emerge a Tasmanian dialect along with Northern Queensland dialects, where the layout (not only keywords and

delimeters) varied, provided the deep structure was the same. The problem for testing thus calls for something completely new: it is the deep structure and the correctness of the syntax tables that need to be tested, not prose forms of a particular dialect. Or alternatively, the validation suite has to be capable of regenerating itself in local dialect form: the translator box beloved of science fiction...

7

The Pascal Standard from the implementor's viewpoint

J. Welsh and A. Hay

Introduction

The Pascal Standard [1] presents a new challenge to implementors, a challenge that is reinforced by the Pascal Validation Suite. When compared with the previous Pascal Report [2] the length and detail of the new Standard must give an implementor cause for concern. This apprehension can only deepen at the prospect of 400 or so programs specifically designed to find shortcomings in any compiler. What then lies in store for an implementor who takes up this challenge?

This paper summarises one implementation project's experience to date in meeting the requirements of the Standard. The project began out of technical interest during the development of the draft standard itself, but has now become an NPL supported project to provide:

(1) a Standard Pascal static checker (SPSC) that will enforce all compile-time enforceable checks on Pascal programs, and

(2) a Standard Pascal model implementation (SPMI) which will demonstrate the techniques required for implementation of all run-time error checks implied by the Standard, albeit on an idealised P-machine architecture.

Both the SPSC and SPMI will be made available by NPL and BSI to help and encourage other implementors to achieve a high level of conformance to the Standard and the Validation Suite. At the time of writing, the SPSC is complete, in the sense that it passes all relevant tests in version 3.0 of the Validation Suite. The SPMI should be complete by mid-1982. The development of the SPSC and SPMI is based on considerable previous experience of implementing Pascal. In the case of the first author, this dates from the initial bootstrap of a Pascal compiler to an ICL 1900 at the Queen's University of Belfast in 1971 [3]. Subsequent developments in Belfast led to an improved modular structure for Pascal compilers [4], a systematic method for exploiting compile-time range analysis to avoid the need for run-time error checks [5], and, in conjunction with the University of Glasgow, a source-related run-time diagnostic package [6]. The resultant 1900 Pascal system has been used as the basis for several other Pascal implementations, and its distinctive features have been incorporated and refined in the SPSC/SPMI development. In the case of the second author, previous experience includes a P-code implementation and maintenance of the highly successful Pascal 6000 compiler for CDC computers.

Conformant arrays

Conformant array parameters are undoubtedly the major addition to Pascal

from the implementor's point of view. Although strictly an optional feature, their definition within the Standard makes their implementation mandatory for all serious implementations of the language.

Handling variable length array parameters is not a new problem and the basic techniques have been well established in other languages. Indeed, some Pascal implementations, such as the Pascal 6000 compiler [7], already offer similar facilities with a somewhat different syntax. However, the Pascal Standard's equivalence rules for array types (which equate multidimensional arrays (one dimensional) arrays of arrays of etc.) do create some traps for the unwary implementor. Suppose we have declarations, as follows:

```
type vector = array [1..3] of real ;
var vmatrix : array [1..10, 1..20] of vector ;
```

The entire variable **vmatrix** may be passed as parameter to a procedure P declared as:

```
procedure P (var a: array [m1..n1: integer; m2..n2: integer]
            of vector)
```

Alternatively, a single row of **vmatrix**, **vmatrix[i]** say, may be passed as parameter to a procedure Q declared as:

```
procedure Q (var b: array [m3..n3: integer] of vector)
```

Within Q the parameter b may in turn be passed as a parameter to a procedure R declared as as:

```
procedure R (var c: array [m4..n4: integer; m5..n5: integer]
            of real)
```

Because of this multi-level view of array structures in Pascal, each bound pair required to describe an actual conformant array parameter must be constructed or copied independently. Because several actual array parameters may share the same set of bound pairs, the base address for each must also be passed and stored separately from the bound pairs. Within the receiving procedure, the set of bound pairs is used in conjunction with the approximate base address to provide an effective array descriptor. For these reasons, the copying of fixed-format array descriptors is not an adequate means of handling Pascal's conformant array parameters.

The Standard's definition of value conformant arrays is carefully worded to allow the necessary copy of the actual array parameter to be made in the calling environment where its size is known, so enabling fixed size procedure activation records to be maintained. In practice, however, implementations that use extensible activation records may find it more convenient to create the copy in the called procedure after control has been transferred, so avoiding any interaction between any copies created and further evaluation of the parameter list, and any problems with their subsequent disposal.

Provided the implications of the Standard are understood, the implementation of the conformant array parameter mechanism reduces to a

straightforward if painstaking encoding of the Standard's carefully worded rules. In the static checker the analysis required for conformant arrays forms an alternative parallel path to that for handling fixed arrays and constitutes an approximate 7% increase in the checker's overall length. The corresponding logic to generate conformant array access code in the SPMI is not yet complete but is expected to produce a similar increase in the overall compiler size.

The relative speed of element access for conformant and fixed arrays depends on the target machine involved and the indexing techniques used on it. For some, conformant array access will be significantly slower, for others it will not. With the mechanism used in the Pascal 6000 compiler access to 2-dimensional conformant array elements is about 20% slower than that for equivalent fixed arrays. What is generally true is that elimination of subscript checking by compile-time range analysis as described in [5] is not immediately applicable to conformant arrays, since the necessary subscript bounds are not known at compile-time. Thus, the considerable overheads normally associated with subscript checking in FORTRAN or Algol will remain for conformant arrays in Pascal. The development of some means of eliminating such checks, at least in simple contexts such as :

```
procedure P (var a : array [m..n:integer] of T)
   var  i : integer ;
   begin
     for i := m to n do .... a[i] .....
   end ;
```

is an obvious priority for compilers supporting conformant arrays, which we hope to pursue in a second phase of SPMI development.

Additional compile-time requirements

The Standard imposes a number of requirements on Pascal implementations which involve additional compile-time processing, but no change in the executable object programs produced. These arise mainly from additional or more precise definition of language features to enable more compile-time detection of programming errors, but in one case from a greater freedom of expression now permitted than that previously.

Formal procedures and functions

A significant syntactic extension introduced by the Standard is the notation required to define the parameter list requirements of procedures and functions which are themselves formal parameters of other procedures and functions. At first sight, this additional feature appears to require significant additional syntax analysis code within a compiler, but in practice this is not so. Because the declaration of a formal procedure or function takes the same form as a heading used in actual procedure and function declarations, a well-structured compiler will use the same analysis code for both - giving rise to a reduction in the overall code required rather than an increase.

Identifiers of arbitrary length

The Standard now makes clear that Pascal identifiers may be of any length and that all characters of identifiers are significant in distinguishing between them. In the past, most compilers have followed the recommendation of the Report in limiting significant length to 8 or 10 characters, and, therefore, have limited the storage required for identifier spellings at compile-time. For these compilers the new Standard demands a change, which should be implemented to achieve the following objectives:

(i) to avoid any time overhead in distinguishing identifiers from word symbols,

(ii) to minimise the time overhead in comparing identifiers, and,

(iii) to minimise the storage overhead in holding the full identifier spellings.

In the SPSC these objectives have been realised by representing identifier spellings as records of type alfa defined as follows:

```
alfahead = packed array [1..headlength] of char ;
alfatail =  tailrecord ;
tailrecord = record
               chunk : packed array [1..chunklength] of char ;
               rest : alfatail
             end ;
alfa = record
         head : alfahead ;
         tail : alfatail
       end
```

The headlength is chosen to be greater than the length of the longest word-symbol (**procedure**), so that identifiers and word symbols may be distinguished by direct comparison of heads only, and in this way objective (i) is achieved. To achieve objective (ii) identifier comparison during table searching uses an in-line comparison of heads, followed by a function call to compare tails if necessary. Because the vast majority of comparisons are resolved by the head comparison alone the resultant degradation of table searching speed is negligible.

The length of chunks used within identifier tails should be chosen to give an economic balance on the particular computer involved between the chunk storage itself and the overhead of their linking pointers. In practice, reasonable variations in this balance have little impact on the overall storage requirement as the majority of identifiers used by Pascal programmers generate little or no tail storage. The major overhead in meeting the requirement of the new Standard is, therefore, the additional tail pointer (usually **nil**) that must be stored for every identifier. For many current implementations this represents a storage overhead of around 10% per identifier table entry. Since identifier storage is a significant proportion of the total storage requirement in compiling large programs, (25% for the 1900 Pascal compiler itself), the new requirement will produce a measurable storage increase for compiling such programs.

Identifier Scope Rules

The rules of scope in the new Standard now make clear that a use of a non-local definition of an identifier in a block B (or any nested block) must not precede its redefinition in block B.

Thus, in the following fragment both N and X are redefined illegally within procedure P:

```
const N = 10 ;
      X = 100 ;
procedure P ;
   type T = 1..N ;                    {using constant N}
   var N : integer ;                  {redefining N}
   procedure Q ;
       begin
         write (X)                    {using constant X within Q}
       end {a} ;
   procedure X ;                      {redefining X in P}
       .
       .
```

In the past, most Pascal compilers allowed such programs by considering the scope of a new definition to start from its defining point – an easy implementation option. Sale [8] has described a simple algorithm for enforcing the new rules but it involves numbering all scopes encountered in the program in order of their opening, and recording in each identifier table entry the number of the latest scope in which it is used. The algorithm produces no significant overhead in compilation-time but does mean the storage of one new field in each identifier table entry. The precise impact of this additional field on the total storage will depend on the limit set for the total number of scopes in any program, and on the packing possibilities within the existing identifier table entries, but in general it may lead to a measurable increase in table storage of the order of that described for the implementation of identifiers of arbitrary length.

For-statement restrictions

The Standard imposes new restrictions on the control variable of a for-statement which makes illegal any assigning reference that threatens to alter the value of the control variable within the body of the for-statement, or within any procedure or function declared in the same block. To enforce these rules the compiler must create and examine two sets of variables for each block B:

(i) the set T_B of local variables that are threatened by any local procedure or function,

(ii) the set C_B of local variables currently in use as control variables of for-statements.

The set T_B is constructed during compilation of the nested procedures and functions of B, and examined during compilation of the statement part of B, so it exists throughout the compilation of B. The set C_B exists only

during compilation of B's statement part.

Each set may be represented either as a corresponding boolean flag held in the table entry of each variable identifier, or as a chained list of pointers to relevant table entries. If the additional boolean flags can be accommodated within existing packed identifier table entries without increasing the overall entry size the first method is preferable, otherwise the second may be more economic. When using the latter it should be remembered that only those variables that can occur as control variables, i.e. those of ordinal type, need to be recorded in the threatened set T of any block.

The processing required on meeting any assigning reference to an entire variable v declared in block B to be of ordinal type is as follows:

```
if B is not the local block
then T_B := T_B + [v]
else if v in C_B
     then error {assigning control variable
                 within for statement}
```

On encountering a **for** statement for v := .... the compile-time processing required is:

```
if v is not of ordinal type
then {control variable must be of ordinal type}
else if v is not local to B
     then error {control variable must be local}
     else if v in T_B
               then error {control variable is threatened
                           by procedure or function}

     else if v in C_B
               then error {same control variable in nested fors}
               else C_B := C_B + [v]
```

and at the end of the statement the processing required is simply:

```
C_B := C_B - [v]
```

With either of the set representations suggested, implementation of this logic does not result in a significant increase in the compiler code length or in the compilation time for typical Pascal programs. Measurements with the SPSC indicate that even with the chained representation of identifier sets the storage overhead for maintaining the sets is not significant.

Label/Goto Restrictions

The Standard requires the enforcement of checks on the accessibility of labels by **goto** statements, with an additional specific restriction on the non-local **goto** statements allowed.

To enforce the checks each declared is represented by a record defined as follows:

```
labeldepth = 1..maxint ;
labelrec = record
             ..... ;
             case sited : boolean of
               false : (maxdepth : labeldepth) ;
               true  : (case accessible : boolean of
                          false : ( ) ;
                          true  : (depth  :  labeldepth))
           end
```

At the declaration of a label its record L is initialised as follows:

```
with L do
begin
  sited := false ;
  maxdepth := maxint
end
```

At an occurrence of a **goto** statement that references a label with record L the checking required is as follows:

```
with L do
  if L is non local
  then maxdepth := 1
  else if sited
       then begin
              if not accessible then error {label too deep
                                            this goto}
            end
       else if maxdepth > depthnow
            then maxdepth := depthnow
```

At an occurence of a statement labelled with the label with record L, the checking code required is

```
with L do
  if sited
  then error {doubly sited label}
  else begin
         if maxdepth < depthnow
         then error {label too deep for previous goto} ;
         sited := true ;
         accessible := true ;
         depth := depthnow
       end
```

The variable depthnow is initialised to 1 at the start of each statement part, and is increased by 1 at the start of each unlabelled statement, i.e. *after* processing the statement label, if any, as above. At the end of each statement the following processing is applied to *all* label records for the current block.

```
for all label records L do
  with L do
    if sited
    then begin
```

```
            if accessible
            then if depth = depthnow
                 then accessible := false
          end
      else if maxdepth = depthnow
           then maxdepth := maxdepth-1 ;
  depthnow := depthnow-1
```

The cost of this label processing at the end of each statement is insignificant only because the number of labels declared in any Pascal block is usually very small, if not zero.

Additional run-time requirements

Apart from its introduction of conformant array parameters, the Standard impinges on the run-time behaviour of Pascal programs in two ways:

(1) A number of minor adjustments and clarifications of the executable effect of existing Pascal features may require corresponding changes in the generated code or run-time support routines of existing Pascal implementations. The changes required are clearly dependent on the existing implementation concerned and cannot be catalogued in any general way, but typical examples are the code generated for the **div** and **mod** operators, or the conversion routines used for numeric output.

(2) The Standard now provides a precise definition of those run-time events that are designated errors and should be treated as such. Although compliance with the Standard does not **require** the detection of errors at run-time, their clear definition now makes it possible, and highly desirable, for implementors to provide such detection, at least as a user option.

Appendix D of the Standard catalogues 59 errors whose occurrence at run-time should be detected. In the following sections we summarise the problems and run-time cost of detecting these errors under six general headings.

File errors

Of the 59 errors listed, 14 relate to the state of file variables during operations on them, or the values of parameters involved in such operations. None of these errors is difficult to detect within the file processing code involved, and most are already handled by existing implementations or require only minor adjustment of these. Because of the nature of file processing and the way it is implemented, the cost of implementing the detection of these errors is not significant, either within the compiler or within the programs generated. There seems no justification for an implementation that fails to provide such detection, or for providing it on an optional basis.

Range errors

Almost half the errors listed (29) relate to limits on the range values taken by scalar variables and expressions in certain contexts. These include the

familiar array subscript error, case index error and subrange variable assignment error, but also include overflow during real or integer arithmetic, which are also range variations of an implementation-defined kind. These errors have always been well-defined in Pascal, and their detection is supported by most implementations, at least on an optional basis. The significant feature of Pascal in comparison with other languages is that the careful use of subrange declarations can enable the compile-time elimination of run-time range checks in many cases [5], and so avoid the high run-time overheads traditionally associated with range checking. In the case of the 1900 Pascal compiler the compile-time analysis to achieve this elimination of run-time check led to a 3% increase in compiler code with a decrease in compilation speed of less than 1%. Such costs are negligible when compared to the benefits obtained, and the adoption of similar range analysis techniques in other Pascal compilers is strongly recommended. The difficulties in exploiting the technique for conformant array indices does not diminish its effectiveness in other areas.

Undefined values

Six errors are listed which involve the use of undefined (i.e. unassigned) variables or function results. These errors have always been clearly recognised, but few implementations have attempted to detect them because of the prohibitive run-time overheads involved.

In principle, each possibly undefined variable v of type T must carry with it a defined tag, if undefined value errors are to be detected. In effect, the required representation has the form :

```
v : record
      case defined : boolean of
       false : ( ) ;
       true : (value : T)
    end
```

The defined tag is set false at the creation of v, becomes true when v is assigned a value, and must be checked in each context where v is required to have a defined value.

In practice, the defined tag can often be accommodated within the representation of the value itself without additional storage overhead, but the high cost of setting up the undefined value representation and of checking for it subsequently still remain.

For variables of structured types the error detecting requirements are even more demanding. Since each component of a variable of (unpacked) structures type may be passed as a variable parameter and manipulated without any awareness of its surrounding structure, it follows that each component must carry it own defined tag with it. This in turn means that a (partially) undefined test for a structured variable involves inspecting the defined tags of each of its components in turn. For variables of a packed structured type, the components are not independently accessible, and for these an alternative strategy of grouping defined tags together for easier inspection could be considered. It is not clear, however, that in general the advantages of doing so would outweigh the disadvantages for individual

component updating.

Some reduction of the high run-time cost of undefined checks may be achieved at compile-time, by maintaining a record of the accessible variables in any block that have definitely been assigned values at or since block entry. For value parameters, and for simple local variables which are often assigned values before any conditional control statements are encountered, this technique should enable the elimination of the corresponding undefined checks, with a significant reduction in the code storage overheads incurred. Unfortunately, however, the highest time overheads result from checks applied during repetitive processing, usually of data accessed either by array indexing or via pointers. The dynamic nature of this access often precludes elimination of the corresponding checks by any simple compile-time analysis. In such cases, hardware assistance in implementing the checks is the most likely means of reducing the overheads involved.

Variant errors

The Standard provides a precise definition of the rules governing variant records with and without tag fields. These rules give rise to six detectable errors that occur through illegal variant field manipulation or through inconsistent use of the extended form of new and dispose. The checks for these errors are the most complex of those required by the Standard, both in the compile-time processing required for their generation and in the run-time processing that their application involves. A tentative model for their implementation within the SPMI has been defined, but it would be foolhardy to describe it in detail in advance of its implementation and testing. Its essential ingredients are the maintenance of a variant check record for each active level of variant nesting within each record together with a cascade of variant checks, from the outermost variant level inwards, to establish the validity of variant manipulaton at any level.

Whatever the precise form of the checks involved, it is clear that their introduction will put considerable overheads on the processng of variant record data. The scale of these overheads makes it imperative that compile-time techniques for the elimination of some of these run-time checks be investigated. The with-statement in Pascal gives a basis for a limited reduction in such checks - by propagating local assertions about the currently established variant status of the record within the with-statement it should be possible in suitable cases to perform the necessary variant checks once per with-statement rather than once per variant field access. If a case-statement with the tag field as case-index is used within the with-statement it may be possible to eliminate the checks altogether. In general, however, the dynamic access (via pointers) used for many variant records will make elimination of variant checks difficult to achieve on any global basis.

Pointer errors

The new Standard lists only two errors specifically relating to pointers, other than those concerning undefined values or the creation and disposal of variant records. These errors involve the occurrence of the special pointer value nil in an identified-variable access or in a dispose operation, and as

such are easily detected.

However, the undefined value error for pointers is a more significant problem than it is for variables of other types. The Standard states than when an identified variable is disposed all pointer variables pointing to it become undefined, but it is impractical for an implementation to locate such variables and adjust their values at that point. Thus 'dangling' pointers are created whose subsequent use must be detected and reported as an undefined pointer error.

To implement these checks the technique described in [9] seems the logical choice. This relies on the incorporation in each pointer value created by a new operation of a unique key value which is also stored within the identified variable storage allocated. The representation of a pointer is thus equivalent to that of the following record type :

```
pointer = record
              case defined : boolean of
             false : ( )
             true : (case nilvalue : boolean of
                     true : ( ) ;
                     false : (key : keyvalue ;
                              address : heapaddress))
          end ;
```

and the representation at every heap variable allocated has the form :

```
record
   key : keyvalue ;
   variable : ......
end
```

The check required for each identified variable p↑, and for each dispose(p) operation, is then as follows:

```
with p do
  if not defined
  then error {use of undefined pointer}
  else if nilvalue
       then error {use of nil pointer}
       else if p.key <> p↑.key
            then error {use of dangling pointer}
            else.....
```

Since the key value allocated by each call of new is unique and the stored copy of it in a disposed variable is destroyed in the dispose operation, the above logic is sufficient to detect most dangling pointer errors. It is not totally secure because subsequent re-partitioning and re-use of the storage addressed by a dangling pointer may accidentally recreate the exact bit pattern of the original key! In addition, if the storage set aside for keys within pointers and heap variables is to be fixed in size, some limit must be imposed on the number of unique key values available, and hence on the number of new operations that can be carried out by any program. Such a limit, however large, may be unacceptable for some application programs. With these provisos, the unique key technique provides a practical means of detecting

the vast majority of dangling pointer errors.

The cost of achieving this pointer security is significant, in terms of both run-time storage and execution speed, and all means of reducing it must be considered. Unfortunately, the nature of pointer processing makes it difficult to achieve much reduction by additional compile-time analysis. Even in programs areas of passive pointer inspection, involving no modification of the data structures represented by pointers, it is difficult for a compiler to establish that any pointer value is 'safe' and not in need of checks. If significant reductions in the cost of checking Pascal pointers are to be achieved it is likely to be by hardware assistance that makes the checks more efficient, rather than by compile-time analysis that eliminates the need for them.

Existence errors

The Standard requires that a variable must continue to exist as long as any reference to it exists. Many references are transient and pose no problems in this respect, but references of extended duration do arise from passing a variable as a variable parameter, or from a with-statement in the case of a record variable. Assignment statements and even indexed variables, may also give rise to extended references depending on the implementation chosen.

Appendix D of the Standard explicitly identifies two ways in which this requirement may be violated - by performing a file operation when a reference to the file-buffer exists and by disposing of a heap variable when a reference to it exists. However, the same error may arise through a change of variant in a variant record and is a special case of a more general variant record error.

To detect these errors requires the association of a reference history with each file-buffer variable, heap-variable and variant part. This history must be updated when an extended reference is established and checked when a file operation, dispose or change of variant occurs. Resetting the history must take place on leaving the procedure, with-statement or assignment that established the reference, either normally or abnormally via a **goto** statement. To do so seems to require the maintenance of a stack of extended references at run-time. Code to unwind this stack (to compile-time determined levels) must be inserted at all points where the history may have to be reset, including all statement labels that may be used for abnormal exit from procedures or statements.

A precise moel for the implementation of these checks has not yet been defined, but for straightforward one-pass compilers, it appears that they will impose some storage overhead on all variant records, heap variables and file buffers, and an additional overhead, in storage and time, on many procedure calls. The extent to which these overheads can be avoided by compile-time analysis requires further investigation.

Conclusions

Experience to date on the SPSC/SPMI project suggests that the new Pascal

Standard can be fully complied with, both in its mandatory compile-time requirements and in its optional run-time error detection. The new conformant array facility is a significant addition for many existing implementations, but is clearly implementable at a reasonable cost. The other additional compile-time requirements are easily met, and the cost of doing so is significant only in the increased symbol table storage that may result. The additional run-time checks, some of which were not previously well defined, are also implementable, but at a significant cost in the running time of the resultant programs. The benefits of retaining such checks in running programs, not just during development but throughout the programs' useful lifetime, more than justify further investigation of techniques to reduce the cost involved by some judicious blend of additional compile-time analysis and run-time hardware assistance.

References

[1] Specification for the Computer Programming Language Pascal, ISO 7185.

[2] Jensen K and Wirth N, 'Pascal-User Manual and Report', Lecture notes in Computer Science, 18, Springer-Verlag (1974).

[3] Welsh J and Quinn C, 'A Pascal Compiler for ICL 1900 Series Computers', Software - Practice and Experience, 2 Vol. 1, 73-77, 1972.

[4] Welsh J, 'Two ICL 1900 Pascal Compilers', in Pascal - the language and its implementation (ed. D W Barron), John Wiley & Sons, 1981.

[5] Welsh J, 'Economic Range Checks in Pascal', Software - Practice and Experience, 8, 85-97, 1978.

[6] Watt D A and Findlay W, 'A Pascal Diagnostics System', in Pascal - the language and its implementation (ed. D W Barron), John Wiley & Sons, 1981..

[7] Strait J P and Mickel A B, 'Pascal 6000 Release 3', University Computer Centre, University of Minnesota, Minneapolis, USA (1979).

[8] Sale A H J, 'A Note on Scope, One-Pass Compilers and Pascal', Australian Computer Science Communications, 1, 1, 80-82, 1979.

[9] Fischer C N and LeBlanc R J, 'The Implementation of Run-time Diagnostics in Pascal, IEEE Transactions on Software Engineering, 6,4, 313-319, 1980.

8

A manufacturer's view

B. A. Byrne

A manufacturer may be interested in Pascal for at least two reasons. The first reason is for systems and applications programming within the company, the second reason is to offer Pascal as a product line item for customers.

Looking at the first reason in the case of a manufacturer using several different kinds of hardware, we immediately recognise the ideal requirement: a program once written should run unchanged on all machines of the range. In practice, of course, this may not be totally achievable because of word-length and other environmental considerations, but certainly to minimise portability problems within the company the requirement will be that there is a standard language available on, or for, all models of hardware. If the further need is identified to be able to sell the source of the manufacturer's programs to third parties or buy in such source from outside, the language standard can only sensibly be the international Standard.

Similar considerations will apply to customers, they will want the assurance, for instance, that all published algorithms in Pascal will run on the implementation they have purchased and that standard training courses and published text books will provide suitable education for their programmers. The existence of a standard will give rise to contractual certainty: customers will know exactly what it is they are ordering; suppliers will know exactly what it is they are required to provide. Thus there is a rational ecomonic argument for a standard, but the prudent manufacturer is sensitive to the market which though Gresham's Law (the bad drives out the good) or some other invisible hand arrives at a standard – the so called de facto standard – that is not the international one. The people that pay the money determine the product or, to put the matter in a more acceptable way, the final ratification of any international language standard comes from the whole computing community.

In the late sixties, among people working on FORTRAN compilers, the general attitude seemed to be that we were doing the user a favour by extending the language and there was a sort of mild competition among suppliers to provide the most attractive extensions. At the same time, a sort of folk-lore grew up that was at least as powerful as the ANSI standards. For instance, it was believed that users expected the values in local variables to be preserved over sub-program calls and considerable effort was put into this despite the fact that no standard required it.

ALGOL was a special case because the original Standard left it to the implementor to decide how something as fundamental as I/O was to be provided.

However with the growth of networks, and because of other ways of providing multi-machine environments, non-standard extensions will probably be seen in many cases as a time-consuming annoyance. Extensions may have

their place, but it should be possible always to confine working to the Standard mode.

The justly popular Pascal language has a Standard in many ways superior to those of earlier languages. Concepts such as conformance and deviance are given, at last, a definition that permits and indeed encourages validation. The intention must be to make ISO Pascal available to Pascal users.

Given that we have this Standard, the next question is how we go about verifying, for any particular Pascal processor, whether or not it agrees with the Standard. ISO DIS 7185 rigorously defines compliance in section five but sets out no means by which compliance could be checked or disputes settled.

To remedy this, the British Standards Institution will soon issue certification for Pascal processors that pass a suite of test programs specified jointly by the National Physical Laboratory and the University of Tasmania. But who validates the validators? Well constructed by well-informed people (*1) though the tests may be, these tests may still contain errors and may not be exhaustive, may not, in principle, even be capable of being exhaustive. It is agreed already that the test suite does not provide an authoritative interpretation of the Standard. Thus though the suite, through it successive iterations, may converge to correctness, it will remain open to a sufficiently determined implementor to avoid certification and still make out some sort of case that his interpretation complies with the Standard. We should aim for a climate of opinion in which certification is seen for all practical purposes as the guarantee of compliance. This, as I understand it, is the purpose of this conference, to establish the place of the tests in the minds of Pascal users and implementors.

Having dealt with these general points, I would like to say something about our use of the NPL test suite within the Company (ICL). The NPL suite consists of approximately 400 programs, many are clever and some, such as the ones that test side effects and the one that determines the properties of real numbers on the target machine, are ingenious. All but an insignificant number of these programs are in correct, Standard Pascal. This in itself means an immense saving to implementors who typically find more errors in their tests than in the product. The tests are short, testing usually only one feature so that it is generally easy to understand what they are trying to do and, more to the point, whether they have done it. In four hundred tests, there is impressive coverage of the language and the features expected in a standard processor. The whole suite is easy to run and easy to check. It is as far as development is concerned, a great and reliable time-saver. We have run and checked all these tests, including rebuilding the compilation system twice, in one and a half working days.

For various reasons, we have currently four different Pascal implementations: there is PERQ Pascal, developed by the Three Rivers Corporation, and bought in with the product in a recent venture and about which I propose to say little since we have taken no formal part in the software development. In a previous collaborative venture, our company acquired the 2900 Pascal compiler and so it was born in those halcyon days

---

(*1) In fact, the tests are written by the same people who wrote the Standard.

before the BSI language Standard was dreamt of.

At about the same time we recognised the need for an internal implementation language that would provide portable, reliable software. For the short to medium term, it was decided that a Pascal conforming to the Standard, but with minimal extensions, was the language to aim for, and the chosen system was a main-frame based cross compiler designed so that code generators for different target machines, typically microprocessors, could be quickly and efficiently produced.

A feature of this system as it now stands is that there is a built-in language level switch covering conformance levels zero and one and then opening up our language extensions.

From this implementation there sprung the DRS Pascal system. This has been extensively reworked to become an interpretive, hosted system running in a microprocessor RAM node that it shares with the local part of the operating system, but with a similar language commitment to the ISO standard.

We have used the NPL test suites versions 2.2 and 3.0 with the last three of these systems; for the more mature Southampton compiler our intention was the honest one of comparing the language level with the Standard. Even here, on what should have been an easy run, we found difficulties. The implementors of this compiler has interpreted Jensen and Wirth as demanding that there should be no semi-colon before **end** in a record declaration. This single quirk meant that about half a dozen tests, meant to examine other features entirely, did not give the intended results. Having removed this trifling anomaly, however, we were able to use the tests as a yardstick with which to measure the implementation against the standard. With the cross-compilation system, the one designed to be standard, the initial view was that the tests would provide sufficiently powerful insecticide to drive out the bugs from the developing product. The tests were not designed for this purpose and although some success was achieved in this respect it now seems naive to have held this expectation. Two gaps in version 2.2 were found in a negative sense by this method. There were no tests for non-text I/O, except for one that wrote to a file and never attempted to read the file in again. Similarly, the tests on set-handling were none too stringent. The exercise of compiling the compiler through itself gave us greater confidence in the robustness of the product. In fact, if there is one overall advantage and disadvantage of the test programs it is that they are short. One is led to the uncomfortable feeling that a processor might pass all these individual tests and still fall over when it saw its first real program.

An obvious difficulty here is that if one wished to systematically investigate, say, combination effects in a language with N features, pairwise testing would require of the order of N^2 tests. And it gets worse for more complicated effects, so there is a limit to what can realistically be achieved. A processor that achieves certification is not, for at the current state of software engineering that is more than we can expect, a bug-free processor but it is a complying processor.

A feature of the tests that was annoying in the context of developing a language processor was that some tests contained unnecessary language features. For instance, `real` declarations where no test is made of floating

point operations. **Real** was given a low development priority in an implementation designed originally for systems programming. The tests also showed unnecessary duplication. For instance, there were three tests to check whether a **for** loop control variable was an entire local variable. With version 3, the test suite heralds in a more scientific age, providing a step towards the automatic checking that BSI will provide with version 3.1. In general, this suite of tests has given a soberly objective account of our implementation, giving us (after allowance for erroneous tests) a conformance score of 100%, and a similarly high score on the deviance tests.

The tests themselves are sometimes erroneous and are still not exhaustive. My colleague, Andrew Williams has prepared a complete report [1] of this, but I mention here as examples of errors 6.7.2.2-13 which should be a test of **mod** but in fact tests **div**; 6.6.6.1-1 which tests standard functions as actual parameters, contrary to the Standard; in addition the conformant array tests contain trivial syntax errors and are based on a previous version of the (at that time rapidly changing) standard. The tests are not exhaustive since there are no tests for local files. Most startling of all, 6.6.6.2-11 miscalculates the minimum exponent in floating point numbers; (Brian Wichmann has since resolved this difficulty). To put this in perspective, we have doubts about only ten out of over four hundred tests - that is the tests themselves are over 97.5% correct. In 3.1, this will of course be 100%. One day all software will be like this.

In summary, we list some good points and bad points about the tests

GOOD	BAD
Test numbering based on corresponding section of standard	Standard kept changing
Simple PASS/FAIL message sometimes particular reason for failure	
Well annotated	Few tests erroneous Some duplication
Very extensive	Not exhaustive
Test short	Tests too short - no "robustness" tests.
Extremely inexpensive for labour involved in their production	

In summary, if the NPL tests had not existed it would have been necessary to invent them. Could we have done so well? In short, the tests are excellent value for money. Everyone should have them.

Reference

[1] A.M. Williams, NPL Pascal Validation Suite Report, ICL internal document unpublished.

9

Pascal validation users' guide

L. Morgan

To have an international standard for a programming Language is one thing. To have a suite of test programs to measure how well a given compiler implements the Standard is another. As a user or supplier of a Pascal compiler, your interest is in how you can take advantage of those facilities. Suppliers will want to use the test suite so that they can bring their compilers into line with the Standard, and have some public recognition that they have done so. Users will want to know which Pascal compilers have been validated and how they fared. To those whose interest is in seeing standards used in practice, it is important that validation facilities are made available in a 'user-friendly' way.

The ultimate aim of the joint project will be the provision of a 'validation service'. At present there is a user guide to running the validation suite. In this paper we will look at that guide and how a future service might operate.

The test suite consists of many programs and running them in a sensible manner and being able to handle the results is not a trivial matter. The user guide leads the user through the steps necessary to run the suite of test programs.

The ingredients of a validation are:

* the test suite;

* the compiler;

* the hardware and software environment in which the compiler runs.

There are a number of steps to be performed to complete a validation. These steps and the relationships between them are presented in Fig.1. It will be seen that some of the activities are independent of one another; thus there is a choice in the ordering of such activities. The user guide is divided into sections so that it may be read in the order in which the tasks are to be performed rather than sequentially.

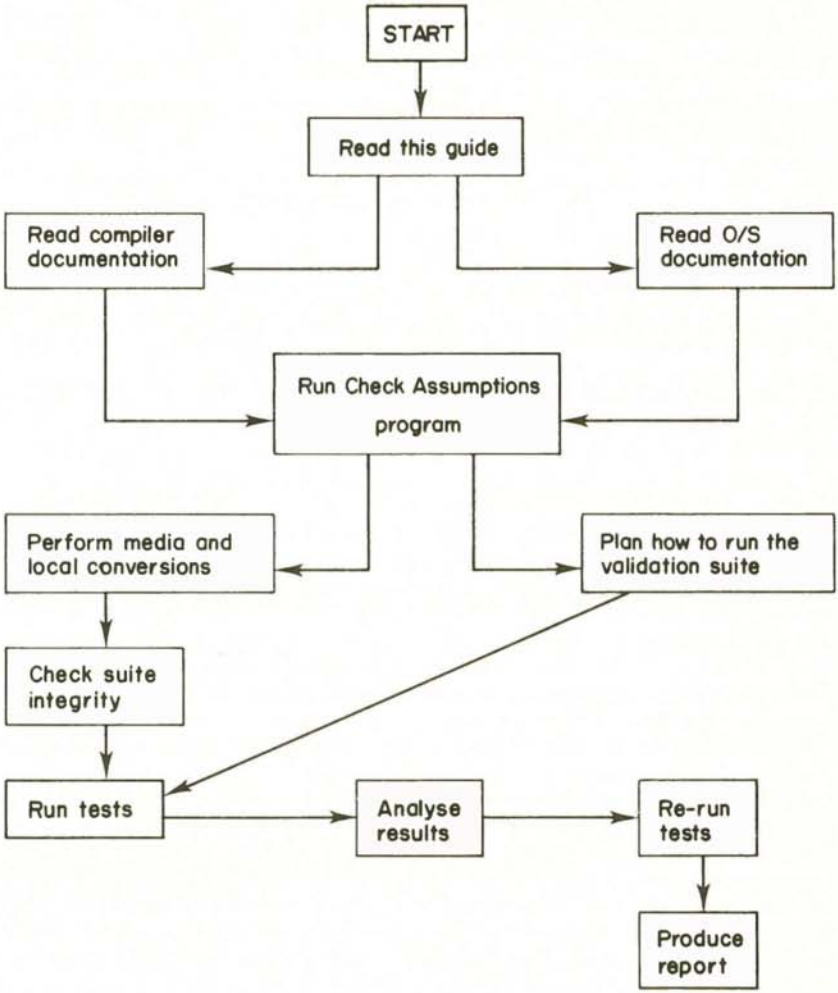
This paper describes the stages in the validation in the order of the user guide, which is:

* Media and Lexical conversion

* Checking the test suite integrity

* Checking assumptions

* Planning the validation run


START
Read this guide
Read compiler documentation
Read O/S documentation
Run Check Assumptions program
Perform media and local conversions
Plan how to run the validation suite
Check suite integrity
Run tests
Analyse results
Re-run tests
Produce report

1.1 Media conversion

The test suite is supplied in the following format:

* half-inch magnetic tape;

* 9-track;

* 800/1600 bpi;

* NRZI/PE;

* even parity;

* 80 character records;

* 15 records/block;

* ISO code;

* unlabelled.

This has to be converted into a format that can be handled by the compiler. This process may involve changes to the character set (Lexical conversion). The Pascal Standard specifies a set of alternative symbols that may be used.

1.2 Checking the integrity

By accident, or design, the source of the test suite could be altered. Thus a special program is provided to check the text by any test program. This is done by producing a 96-bit binary check pattern using an algorithm which was developed for use in data transmission (see Appendix B).

This program must be used before running a test program. It should also be run on its own source code.

1.3 Checking assumptions

In creating the test suite, two sets of assumptions have been made:

1. That certain features of the Language that are requirements of the Standard are implemented by the compiler.

2. That certain details of the Pascal Language and the Pascal compiler have been implemented in a reasonable way.

The validation suite contains a program to test these assumptions for if the assumptions are false then running the test suite is a waste of time (see Appendix C).

1.4 Planning the validation run

Planning the validation process is of considerable importance to the validator. The correct decisions made at this point could result in significant savings of human effort and/or computer resources.

The running of a single test program is achieved in three steps:

1. The test program is extracted from the file containing it.

2. The program integrity is checked.

3. The program is processed using the Pascal compiler which is being validated, and the results produced by the executing program recorded.

These steps must be repeated for the several hundred of test programs. It may be possible to automate some or all of the steps involved depending on the facilities of the host operating system.

1.5 Analysing the results and producing the standardized report

In theory if a compiler fails on a test then it has failed the validation. However, if our aim is to assist bringing compilers into line with the standard the result of a validation must be more than PASS or FAIL.

The analysis of the failures is most important. Simple amendments may allow a fatal error to be by-passed and allow the rest of the tests to run. The final validation report should give

1. The processor validation details from the standardized description.

2. The Level of the Standard to which the validation was performed.

3. The Lexical changes that were needed prior to validation.

4. The results of the validation.

5. A validation summary.

A number of features of the Standard can be implementor defined. This means that the users must be told what these features are, and the compiler must implement exactly what is stated. In order to perform a validation all of these features must be investigated and a 'Standardized compiler Description' produced. This description will also contain information about the make and model of machine, and the operating system used.

2. A Validation Service

In Section 1 we have looked at what needs to be done to perform a validation. How can we expect a future validation service to operate? In this respect we can only look at existing validation services.

The Federal Compiler Testing Center provides a validation service for the

US Federal Government. This service at present covers COBOL and FORTRAN compilers. What follows is based on that service, and gives an idea of how a Pascal service might operate.

The fundamental principle of a validation service is that the requestor of a validation, 'the customer', is responsible for running the validation. The testing centre provides the test suite and user guide, observes the validation and writes a report on the results. Also the customer must pay the test centre for all work involved in the validation.

With many products a stamp of approval, eg the BSI kitemark, can be given. A software product like a compiler does not quite fit that mould. Typically a validated compiler is identified as follows:

* all compilers that have been validated are given a Certificate that signifies that validation has taken place;

* a validation summary report is published that identifies the compiler and all of the problems that occurred during the validation;

* a list of validated compilers plus those scheduled for validation is published.

If a compiler undergoes significant revision (usually the suppliers' work is accepted on this) the compiler must be re-validated. If the test suite is revised then all compilers will need re-validation.

A validation may be requested by a user or a supplier, however, the supplier is informed if a user requests the validation and is also given a copy of the validation report.

Copies of the test suite and the user guide will be on sale to the public.

An integral part of the validation service will be an appeal procedure, which can resolve any problems related to the validation, or to the interpretation of the Standard.

On average FCTC were charging $4000 for a COBOL compiler validation during 1981.

10

The role of BSI in testing

J. W. Charter

1. Certification and Testing

1.1 What is third party certification?

"Certification – The authoritative act of documenting compliance with requirements" (BS 4778 – Glossary of terms used in quality assurance). In addition to being the national standards body for the UK, the British Standards Institution is a prominent third party certification body and a widely recognised testing organisation.

Where a manufacturer has set up his quality management system and looked on it and found it to be good, how does he tell the world? And how is a prospective purchaser to know that the claims of that manufacturer are more reliable than those of some of his competitors? One way to do it is to make use of a third party certification. Look for the Kitemark!

Third party certification has a special role in quality assurance. It may not apply when, for example, a purchaser sets out to buy an expensive, complex, specialised piece of equipment to his own specification. On the other hand, it might very well apply where he wishes to buy catalogued items made to the manufacturer's specification, or to national or international standards.

The presence of a certification mark on a product may not provide a purchaser with all the assurance he requires, but he will at least know that the manufacturer's claim of compliance has the backing of an independent body. This means that he, the purchaser, can relax his vigilance to concentrate on more critical quality areas.

So far as the manufacturer is concerned, in addition to providing support for his claims, third party certification also provides a useful second opinion on his product design, on his interpretation of the standard, and on the effectiveness of his quality control system. These are no small consideration in these days of increasing consumer and environmental protection legislation. And if product liability legislation is ever introduced into the UK, and if compliance with standards is not acceptable as a defence, then manufacturers will still need third party certification to help them to demonstrate that they maintain the highest possible level of compliance with standards, thus keeping the risk of litigation to the minimum.

Of course, Governments are great users of third party certification. The best way to meet the electrical regulations of the provinces of CANADA is to obtain C.S.A. certification. The only way into the Scandinavian market for electrical goods is via the triplets – SEMKO, DEMKO, NEMKO. If you want to sell motor cycle helmets in the UK, you have to deal with BSI. Governments themselves are great third party certifiers. Tractor safety cabs,

insecticides, pressure vessels are only some of the products now certified by UK Government. The 'E' mark of ECE and 'e' mark (and its variations) of the EEC are both in increasing number of products come within scope of mandatory, governmental certification.

So we see that third party certification has a particular place in the scheme of things. It is there to serve manufacturers, purchasers, consumers and governments. Over the years it has developed in various ways to meet these needs.

1.2 The role of testing

"Test - A critical trial or examination of one or more properties or characteristics of a material, product or set of observations" (BS4778).

Third party certification systems are usually made up of some or all of the following elements:

(a) Type testing - an independent and objective assessment of the product against all the requirements of the nominated specification.

(b) Assessment - an independent examination of the manufacturer's quality management system to determine that he has the capability of continuing to produce to the nominated specification.

(c) Surveillance by sampling - the selection of samples from the factory, warehouse or the market for independent testing against the nominated specification to check that the manufacturer continues to comply.

(d) Surveillance by factory visit - the rechecking at intervals that the manufacturer continues to maintain his quality assurance capability.

From this it can be seen that testing in its broadest sense (see the above definition) is the essential element of certification, whether it is the testing of a sample of a product in a laboratory, or the "testing" of the statements in a manufacturer's quality control manual against the reality of what actually happens on the factory floor.

1.3 BSI's Credentials

BSI has been in the certification business for over 70 years and in the testing business for 25 years. It owns a number of certification marks, of which two are currently in use. These are the Kitemark and the so called Safety Mark. Both of these certification systems are based on type testing, factory assessment and surveillance by sampling and factory visits, as described in Section 1.2 of this paper. Some 750 manufacturers hold between them 1400 Licences to use these marks, each licence relating to a range of products produced to a British Standard. Certification schemes are operated to approximately 250 different British Standards.

In 1978 BSI introduced a new form of certification based on the assessment of the capability of a firm to produce goods or supply services within a defined scope of operation. This system fills a gap where for various reasons, product certification is not appropriate - for example, where services are supplied, or where the manufacturer works to codes of practice with the criteria for the product laid down by the purchaser. Currently over 300 firms are registered in a number of manufacturing and service

industries. A new development is that BSI is cooperating with the Ministry of Defence and a growing number of other major purchasing organisations, to operate capability assessment schemes for sub-contractors. These schemes will all have a common base of BS 5750 "Quality Systems", and if recognised by the main contractors to those purchasing organisations, will reduce the multiplicity of assessments carried out on sub-contracting firms.

In testing, BSI offers a range of services far wider than those used for supporting its certification operations. The BSI Test House is, in fact, a separate organisation, commercially based and recognised by a number of Governmental bodies, both in the UK and overseas.

All BSI certification, testing and assessment services operate under the aegis of the Quality Assurance Council, and are financially self-supporting.

2. The Pascal Compiler Validation Project

2.1 Why validation?

As industry, commerce, social services, health services, education etc. come to depend more and more on computers for efficient functioning, so the demand increases for greater reliability of computer performance. Hardware reliability has improved over the years, but it is very doubtful that software reliability has made any significant advances. A fundamental part of computer software is the compiler which for overall reliability of the complete system, must be able to handle programs accurately.

Many compiler producers have developed their own test suites to check that what they are offering to purchasers is working correctly – at least within the bounds of the test suite. For their part, however, certain large user organisations have developed their own testing and acceptance regimes; an activity which has come to be generally known as "validation".

The first point to appreciate about validation is that it is not defined in BS 4778, which means that it is not yet a recognised term in quality assurance. Through custom and use it has come to mean the process of applying a range of tests to a compiler and making statements on its performance as judged from the results. The tests do not together constitute a specification for the compiler, so it is not possible to state whether it has "passed" or "failed" the total test regime. It is for the potential user to judge from the results whether he is prepared to accept the compiler. If he does so, it may be in the knowledge that the compiler has certain identified weaknesses which must be taken into account when it is put into use.

Validation systems of this nature are operated by a number of organisations. The US Government has its Federal Compiler Testing Center for COBOL and FORTRAN compilers. FCTC issues lists of validated compilers and publishes the results of tests. RSRE validates CORAL 66 compilers for the UK Ministry of Defence. What BSI is setting out to do is to determine whether a commercial "third party" validation – or certification – service can be operated for PASCAL compilers, using the test suite developed by the University of Tasmania and NPL.

2.2 Certification or Validation?

It is clear that certification and validation are two different things. The former is a statement on conformity with an identified set of requirements: the latter is a series of observations on performance when measured in a number of identified ways. The question facing BSI, and one of the reasons for the project which is about to commence, is whether it is possible to certify a compiler. If it is possible, is this what the user wants? Or would the market be content with a validation system? Might it be the case that certification is more appropriate for compilers for micro computers and validation for larger computers? Is it possible to certify a compiler without a "specification" to which conformity can be assured?

There is no doubt that whatever the answers to these questions, if the final decision is that there is a need for a PASCAL compiler validation/certification/testing centre in the UK, BSI will be entering into a new phase in the services it provides to industry.

2.3 Particular Problems

Seen through the eyes of someone accustomed to conventional forms of certification and testing, the certification/validation of computer compilers presents some unique problems. For example:

(a) the "product" has no physical boundaries by which it can be identified – therefore how does the purchaser know that what he is buying is the same as that which was tested?

(b) the full range of the performance of a compiler may not be known, even to the person who designed it;

(c) the characteristics of a compiler can only be assessed through the operating system, which itself may contain errors;

(d) some aspects of compiler performance may be untestable in the conventional sense.

These and other factors will be examined carefully during the course of the project. The final decision to launch the project will depend on finding answers to all such questions and on whether the service will be an acceptable financial and legal risk to BSI.

2.4 Programme for the project

It is intended that the project will be of two years maximum duration, and will follow a programme approximately as follows:

First year.

a) Examination of the operation of existing compiler validation/certification systems.

b) Determination of types of validation/certification which it is technically and legally feasible to operate.

c) Determination of market needs through consultation and investigation.

d) Establishment of a working party representative of producers, users and

other interested parties, with which to consult on all stages of the research and development programme.

e) Drafting of rules and procedures for the operation of the system.

f) Operation of "guinea-pig" validations and certifications in cooperation with producers and users; assessment of results.

h) Input of operating experience to BSI Committees for PASCAL standard and compiler testing standard.

i) Examination of the economics of operating projected system; determination of appropriate fee structure.

j) Preparation of "working draft" rules and procedures; approval in principle of Quality Assurance Council.
Second year.

k) Negotiation and development of an introductory programme of validations/certifications on a fee paid basis.

l) Implementation of introductory programme; analysis of results; adjustment of "working draft" rules and procedures.

m) Review, amendment where necessary, and final confirmation of validation/certification carried out under the programme.

n) Final approval of Quality Assurance Council.

o) Promotion and official launch of the system.

3. Benefits of a compiler validation/certification service

The benefits to the computer industry, both producers and users, of establishing an independent service for compiler validation/certification are seen as follows:-

a) Users will be provided with an assurance that compilers attain a known level of quality and uniformity.

b) Producers will have at their disposal an independent authentication of their claims of compiler performances.

c) The system will become a focus of experience for strengthening and improving standard languages and assisting in their development in a disciplined manner.

d) The system will provide a focus for information and expertise for the development and improvement of compilers.

e) The existence of the system will substantially improve the portability of computer software, and will help to reduce software costs.

f) The existence of the system will place the UK amongst those nations

which have developed or are developing compiler validation/certification systems, with consequent benefits to British exporters of computer software.

g) The assurance provided by the system will enable users to better appreciate the limitations of compilers, thus reducing the danger of computer errors.

BSI is determined to bring these benefits to the UK.

11
The SOL project and validation

M. Gien and J. Sidi

Introduction

Software validation has been a great concern in France for several years, but solutions to this difficult problem have not got beyond a few limited experiences yet. The Army which developed the LTR programming language for Real Time applications and had compilers developed for a few computers by various companies required that these compilers be validated. One of its technical centres, the CELAR, designed test programs and checked a few compilers [1]. Most tests were automatically generated, out of a few skeleton programs where complexity was increased by adding to the environment. Running the tests and checking results were tedious because of the huge number of tests, but this was an interesting experience.

An important project called SOL, based on Pascal, has been started recently, with program portability as its main objective. Validation is naturally one of its ingredients.

The SOL project

The pilot project SOL is developing a basic software engineering environment, portable and written in Pascal, intended for supporting research, development and operation of software engineering tools.

This four year project, launched at the end of 1979, is sponsored by Agence de l'Informatique. Project management is carried out by INRIA (Institut National de Recherche en Informatique et Automatique) with participation of CNET (Centre National d'Etudes des Telecommunications). Project participants include research laboratories as well as industrial companies. INRIA, CNET, BNI, CNAM, CERCI, EUROSOFT, SEMA, STERIA, SYSECA, Universities of Paris VI and VII, Toulouse and Tunis are the main participating organizations.

The basic software engineering environment developed in the project includes:

- PASCAL compilers based upon the ISO Standard,
- a portable, Unix-like, standard Operating System,
- a set of basic Software Tools.

The whole environment is portable and written in Pascal. First implementations are intended for 16-bit mini and micro computers including Honeywell/Level 6, Sems/Mitra, Motorola/68000, Intel/8086 and National Semiconductor/16000.

The participation of industrial companies in the project ensures that SOL results will be supplied and maintained on an industrial basis. First products will start being commercially available in 1982.

A validation procedure is being developed in particular for Pascal compilers to ensure that implementations conform in every respect to the standard SOL external specifications. It will certify that adaptations of the SOL environment to various machines are strictly compatible, thus guaranteeing genuine program portability.

The availability of the standard SOL environment on a wide variety of computers, made possible by the portability of the environment itself, aims at constituting a solid basis for Research and Development programs in Software Engineering, leading to portable results.

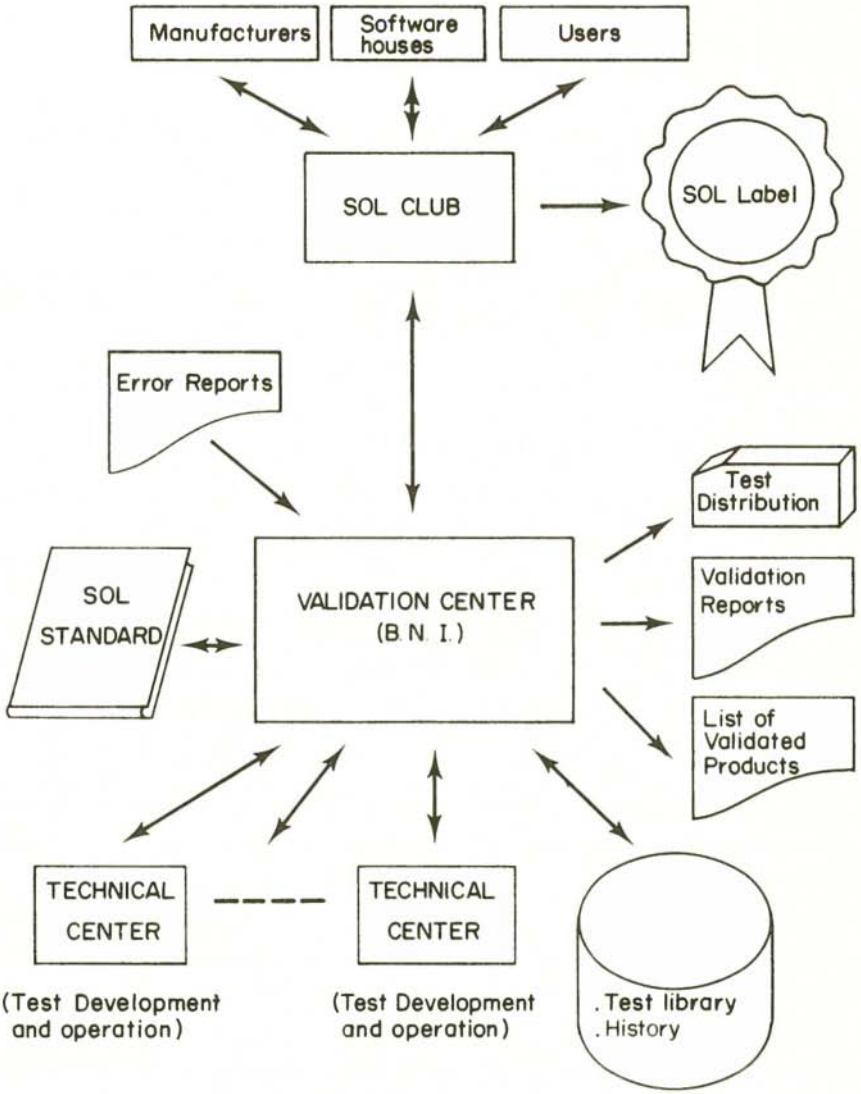
SOL validation

The SOL validation is a fundamental aspect of the effort being made towards program portability. It primarily concerns Pascal compilers but will also be applied to the SOL environment (Operating System and basic utilities).

The SOL Validation Center is sponsored by the SOL Club, an association of users and providers of SOL "products" holder of the SOL standards. The Bureau d'orientation de la Normalisation en Informatique (BNI) is responsible for its organization and operation. Figure 1 shows the distribution of tasks and responsibilities. The validation centre will thus:

- develop, operate and maintain validation programs, with the help of technical centers (INRIA is one of them),
- distribute tests programs to developers in order to help them complying to the Standard,
- participate in the evolution of the Standard and reflect changes in test programs,
- collect error reports on validated products in order to improve test programs,
- propose to the SOL Club the grant of a SOL "Label" to validated products,
- make available lists of validated products.

This organization will first be applied to Pascal compilers, it should then be extended to the SOL system and other software.


Manufacturers
Software houses
Users
SOL CLUB
SOL Label
Error Reports
Test Distribution
SOL STANDARD
VALIDATION CENTER (B.N.I.)
Validation Reports
List of Validated Products
TECHNICAL CENTER
TECHNICAL CENTER
(Test Development and operation)
(Test Development and operation)
. Test library
. History

Pascal compiler validation

Pascal Specifications

The SOL standard for the specifications of the Pascal programming language is defined by the ISO standard specifications [2] completed with SOL specific items, aiming at reducing discrepancies between implementations [3]. SOL additions, for example, define points which are left implementation defined in the ISO standard; they include standard functions and procedures for file access which are usually provided by compilers, as well as standard directives for separate compilation. They try to be as complete as possible to allow the use of Pascal in an industrial environment and to avoid leaving choices to implementors.

Test programs

The "Pascal test suite" produced by NPL and University of Tasmania [4] constitutes, as such, the basis of the SOL validation programs for Pascal compilers. A few tests have been added that behave as "complexity" tests rather than checking a specific point only. Other tests have been developed that correspond to the SOL specific items.

Validation operation

Problems involved by Pascal compiler validation are numerous and not always easy to solve [5].

To run the tests, it is necessary to get access to the computer and operating system which support the compiler to be validated. Therefore, it won't be easy, in the general case, to run the tests in a central place, by the same team. Distant access should be possible though.

The technical center teams need to know the standard specifications, the test programs, operation procedures and result checking. They must be capable, with the help of manufacturer experts, to invoke the functions of all operating systems on which to run the tests.

There are presently 500 test programs. Checking them manually is very tedious. Automatic checking is being studied but a few cases will always be difficult to cope with. Error detection or deviance tests in particular; the abort of the compiler itself or runtime loops are such examples.

Publishing the test programs might help to solve some of these problems, manufacturers being able to "prepare" the validation. It would still be "formally" performed by the validation team but the conditions should be most favourable.

Evolutions

Four elements are essential to compiler validation:

- language specifications,

- test programs,

- compilers,

- compiler running environment (OS and hardware).

Each of these elements is to evolve. Noticing errors on a validated compiler will induce updates in the test programs. A new version of the compiler or operating system may destroy previous validation results. The Standard itself may be modified.

The validation certificate should then be valid only for a limited period of time. Periodic validations would guarantee that any evolution be systematically taken into account.

Validation of the SOL operating system

The SOL operating system [6] is made up of a kernel, providing services necessary for program execution and a set of programs called "utilities" which are essential to the operation environment; some of them like the command language interpreter being very basic. Program portability will be actually achieved between SOL systems if kernel services appear to be identical for every implementation. This should also be true for utilities, though they do not depend so much on hardware and should therefore behave the same as if they have been intentionally modified.

The SOL operating system validation will therefore distinguish between the SOL kernel validation and validation of utilities.

A SOL kernel implementation is realized from a "reference kernel", written in Pascal and implemented on various hardware. Its validation could then be performed by running programs exercising the reference kernel services and comparing results, that should be identical.

Utilities behaviour could be validated in the same way.

Clearly, validation of the SOL operating system is a new experience. It is being studied but it is required in order to control system evolution and avoid future deviations that would surely impair the portability level provided initially.

The organization of this validation should be very similar to what is being set up for the validation of Pascal compilers.

Conclusions

Project SOL is setting up validation procedures for Pascal compilers as well as for a complete operating system. Involving organizations such as a SOL Club and BNI aims at ensuring continuity in that process. These national efforts are very important but they should not be limited to one country. Software is international and so should be validation procedures. The EEC

is sponsoring the validation of ADA compilers which should lead to some coordination, at least in Europe. UK and France are setting up validation procedures for Pascal compilers, they clearly should be closely coupled.

References

[1] LTR, French standard Z 65 - 350, CELAR 35170 BRUZ.

[2] ISO, "Specifications for the Pascal Programming Language", DIS 7185, (Oct. 1981).

[3] P. Maurice, "Complément de spécification à la norme ISO du langage Pascal et mise en oeuvre dans les processeurs Pascal-SOL", SOL Doc. LAN. 1.538.2, (Nov. 1981).

[4] B.A. Wichmann, A.H.J. Sale, "A Pascal processor validation suite", NPL Report CSU 7, (March 80).

[5] M. Kronental, N. Malagardis, "De la Portabilité à la Certification des Compilateurs", Journees d'Etudes Genie Logiciel, Perros-Guirec, (29-30 Jan. 1980).

[6] I. Campbell, P. Chemla, M. Gien, G. Ollivier, "Le Système d'Exploitation SOL", Bul. Liaison INRIA, (Mars 82).

12
Discussion

1. The Validation Suite

Question: When analysing the results, why don't you indicate the reason for a particular test failing?

Ciechanowicz: The only person who can explain the reason for the failure is the author of the compiler. We are involved in third party testing which means that the compiler is treated as a black box. All we can say with confidence is that the test failed. An examination of the compiler output can reveal some interesting information, but it is very labour intensive to record anything other than a simple PASS/FAIL. Also there is a danger of making subjective judgements which cannot be independently supported. It would be useful to make comments on the accuracy of diagnostic messages but this is not done.

Sale: Concerning the failure of certain tests there is an important difficulty with the test suite. The largest class of bugs in the current suite arises as follows. A particular test should fail to compile. It is run on two or more compilers and indeed it does fail to compile. However, the test contains a mistake so that it actually fails for the wrong reason. It is hard to detect these errors and we have not been helped by changes in the various drafts of the Standard.

Wichmann: Please remember that the test suite cannot be considered as being complete. Version 3.1 will be more comprehensive and should have fewer bugs because we now have compilers for testing which more nearly conform to the Standard.

Daessler (Siemens): Concerning version 3.0 of the test suite, I have noticed that all the testing of files is performed with files which are program parameters. What about testing local files?

Sale: The suite makes an implicit assumption that all files are implemented in the same way so that only one form needs to be tested. Version 3.1 will do most of the testing with local files which will avoid the need for additional JCL to run the suite (on many systems).

Question: Should you take legal action against suppliers who supply non-standard compilers?

Wichmann: As part of central government, The National Physical Laboratory would not expect to undertake such legal action. It is up to the market place to settle these issues. In almost all countries there are laws like the UK 'Trade Descriptions Act' which would allow somebody legal redress if a supplier did not comply with his own claims.

Sale: It could be very embarrassing for a supplier if such legal action was successful. He should protect himself by ensuring that the test suite

functions correctly.

Gien: Isn't there a risk that the test suite can get too big by continually adding further tests? Compilers are getting used to some of the tests so shouldn't they be deleted?

Sale: Yes, we try to look at each test to see if it is worthwhile. At least Pascal is not growing as a language and hence the suite does not need to grow on that account. Indeed, one can argue that the introduction of conformant arrays which gives structural compatibility of types is a retrogade step.

Question: How can one test extensions?

Addyman: As far as the Standard is concerned, there is an automatic reviewing process every five years. At that time, the Standard can be reconfirmed, thrown away or revised (presumably with extensions). COBOL and FORTRAN illustrate the extension route of programming languages. Pascal has resisted this so far mainly because agreement was lacking on specific extensions. Also Ada could be seen (if you like it or not) as a very much extended Pascal.

Cichelli: The Pascal extension group in the US has been much more conservative than I expected. In something like 6 meetings over 18 months attended by 100 or so people, the only agreement is for the otherwise clause in the case statement. Rather small progress, although a lot of other proposals have been made. *(Editorial note: Mr Cichelli is responsible for distributing the Pascal validation suite in North America.)*

Sale: Every 3-10 years we throw away a computer architecture and start again. But never in the history of computer science have we thrown away a major language. I think we ought to start learning from the experience with architectures because I don't want FORTRAN to be around too much longer. If in ten years time I am still talking about Pascal I will be disappointed. We can tidy up a few of the implementation problems that Jim Welsh has spoken about. There is only one extension I think is worth doing and that is the deletion of the repeat-until statement. It is a trap and its deletion gives two extra identifiers - hence my calling it an extension.

Addyman: If the Americans have been so unsuccessful in agreeing extensions then it will be even worse with an International group. The conformant array extension already took a large proportion of the total effort.

Sale: As a neutral observer I can confirm how good the US have been on Pascal. Particularly so when one realises that it is a committee of 100. This is the first time that a major programming language has been standardised with the major effort coming from outside the US. The normal rule is for ISO to approve an ANSI standard such as COBOL and FORTRAN.

Question: How long does it take to run the test suite?

Ciechanowicz: On a main-frame it takes about two hours of elapsed time.

Addyman: The problem is that on a microprocessor it is much slower. The temptation is to leave it going unattended but then one test can crash the operating system so that the subsequent tests are not run.

Wichmann: One system had problems with quite a simple test. Many people here will be familiar with the problem of ".." and ":". Replacing .. by : in an array declaration caused the compiler to get into an output loop printing error messages. When the floppy disc was full, the operating system crashed.

Sale: I think the first time you do a validation you need to allow a week depending on how well you know the operating system. The second time is much shorter until eventually it is almost all machine time. The suite nearly always crashes the system once.

Question: Could Professor Welsh say something about the Pascal machine for his compiler?

Welsh: We are using the current Pascal Plus interpreter. This is similar to the UCSD P-code system but includes a number of improvements. The main difference here is the attempt to pass through information to allow better error checking and diagnostic messages. The interpreter itself will be written entirely in Pascal, mainly to give good documentation on what the interpreter is supposed to do.

Obbink (Philips): When can we expect the final approval on the ISO standard for Pascal? Should we start altering our systems now?

Addyman: We are in a very strong position not through good planning and foresight but by some hard work by many people. The final ISO approval would normally wait two years for a French text. But the French text has been available for six months. Hence there is no reason now to delay on the 6-month ballot for the final ISO approval which is usually just a formality. It is inconceivable, based upon past history, for this not to get approval. The ISO standard will consist of an (expensive) reference to the BSI and AFNOR texts of the Standard.

Gien: We actually want to get the French text typeset, so that the fonts have to be settled by following the English text. *(Editorial note: the French text has now been printed).*

Question: What is version 3.1 of the test suite?

Wichmann: This is our name for the next version which has not yet been released. The intention is to remove all bugs in version 3.0, add the tools for the automatic analysis of the test results (to aid BSI) and to add further tests. In fact the tests on conformant arrays have had to be rewritten almost completely, but elsewhere the test suite will change very little. It should be released in September 1982.

Addyman: I would like to ask the audience what they think about the conformant arrays.

Wichmann: There seems to be a consensus in favour.

2. The Validation Service

Wichmann: I'd like to start this discussion about our project by looking at how a validation service might be run. Suppose a supplier wishes to validate his own compiler. He would firstly obtain a copy of the validation suite, and run the tests. Having made his compiler conform to the standard, he would obtain a copy of the guide (mentioned by Lyndon Morgan) and check he could meet all of BSI's requirements. He would then request a formal validation from BSI, BSI would witness the tests, check the results, and in all probability the supplier would end up with a formally validated compiler. However, there are two major problem areas which must be addressed by BSI. Firstly, should BSI merely perform certified testing? Secondly, should the validation reports be made publicly available, or should the supplier have control over their distribution?

Charter: Regarding your first question, perhaps the actual validation could be supported by a capability assessment of the compiler producer. This line has proved profitable in other areas. On your second question it is generally understood in the testing business, that the test report belongs to the person commissioning the tests. However, use of the test report is governed by certain conditions about how it is used. As regards existing compiler validation systems, all the reports are published. Perhaps the only viable system for Pascal that can be operated is one in which reports are made available.

Question: What about the Agrément system?

Charter: Actually the Agrément system is not run by BSI, but by an organisation called the Agrément Board. It assesses and issues assessments of new products in the building industry. Manufacturers pay for the reports which are published.

Gien: Will a certificate be issued only if all tests are passed or will some margin of failure be permitted? If the latter applies then the failures will be publicised.

Sale: I don't see why compilers shouldn't pass all of the conformance tests, but on the other hand, the Standard allows for processors which don't meet all the requirements. So I think that only an assessment certificate will be issued. I'd like to point out that failing to run a correct program isn't too bad, but running a correct program incorrectly is terrible. I'm not sure what we will do about them.

Question: Is it possible for the test reports to include some form of grading?

Sale: Reaching agreement on a grading scheme would be very difficult and that's why reasons for failing particular tests must be included in this certificate. Coming back to the publication of reports, I don't have any strong views on this matter. I think the important thing is that the test suite should remain accessible to any user.

Wichmann: I think that whatever BSI do, we will still distribute the test suite at an extremely cheap price. Its value would be lost if it ever became expensive.

Thompson: From the point of view of the Community, the Commission is trying to break the thing down to two separate parts, that is, we want the professional to assess conformance to the Standard, and we want an assessment of the outcome to come from an external, independent body.

Addyman: I'm in favour of reports being published, although I think it's fair for a supplier to keep the report since he's paid for it. If I can see a list of products that have been tested, and I want to purchase one of them, then I would want to examine the report. If the vendor consequently did not allow me to look at his report, I would naturally refuse to buy his product. That's why I think it's in the interest of vendors to make reports available.

Thompson: Originally the UK MoD published a list of Coral 66 compilers which had been tested and the results of the report were the property of the submitting body.

Slater: That's not true now, both the list and reports are available.

Wichmann: Any supplier like to venture into this area?

Hetherington (Prospero): Any validation based on the current test suite needs a lot of interpretation and I think that a report needs some form of commentary. Secondly, I think the deviance tests have a strong quality flavour about them.

Wichmann: You think that the report should contain cogent reasons for failure of a particular test?

Sale: The important thing is that the test has failed; if you can give a good reason, that helps you get round it. Unfortunately, most people think that conformance tests are more important than deviance. I would certainly put the priorities round the other way.

Hetherington (Prospero): What if a 500-line program is rejected without any explanation?

Cichelli: I've received many reports, since I distribute the suite in America, and they vary from the very sophisticated to very poor. I do keep a list of purchasers of the suite but I haven't made it public. I encourage users to submit test reports to **Pascal News**. Problems with the quality or bias of reports can be offset to some extent by having more than one report on a product. Vendors will never claim too much otherwise their users will complain.

Wichmann: Do you think reports should be more readable?

Cichelli: I'm not sure, since users are becoming more sophisticated.

Sale: As the number of exceptions and things that have to be reported decrease, from the present average of 70 to about zero, I think that interpretation of the report will cease to be a problem.

Charter: Finance might dictate the form this validation service will take. On the one hand, producers might not put up their compilers for validation

if the results must be public, on the other hand, users might not be prepared to seek validated compilers.

Cichelli: If you take the attitude that what we're trying to do in general is to improve the quality of all of the products then you get a sympathetic hearing.

Gien: Once a compiler has been validated and passed the validation, all the others will follow.

Morgan: If suppliers aren't required to publish reports, a user could ask BSI to perform a validation and also publish the results. Then you have a nasty political problem.

Charter: In any certification system you must start off the way you intend to go on. It is tempting to coax the vendors with a cheap and easy system, and once they're committed, alter the system, but it never works. It's imperative to establish the policy before you set up a certification scheme.

Sale: Once we have a validation service I suspect that suppliers will want to validate their products, because if they are sued as a result of a malfunctioning compiler which wasn't validated, then they are in a distinctly poor light. Also, if the customer is shown the report, then that customer can't claim he wasn't shown what deficiencies there were in that compiler. I have spoken to several hundred vendors and implementors and with one exception they welcomed the validation effort.

Wichmann: I've had similar feedback and I am somewhat surprised. Some of the tests are a compiler writer's nightmare because they are so unlike ordinary programs. I think the tests are a challenge for implementors and I am glad the challenge has been accepted.

Gien: Would the Commission be prepared to support an international validation procedure?

Thompson: Basically the member states wish to fund their own efforts, and the role of the Commission is to assemble these separate efforts, rather than funding specific works. So I believe the Commission would fund an internationally recognised testing service.

Wichmann: The idea of issuing validation reports in all community languages is daunting. Negotiating with every national standards organisation as to what status validation reports should have, is also a problem.

Thompson: Ideally all the organisations involved should sign an agreement to resolve any potential differences, determine the procedures which are to be used in a validation, and have a mechanism to resolve any differences between them. It would be a major political step to achieve this harmony on a world-wide level, and I'm sure we would assist in such a process.

Question: Are there any precedents, within the computing sphere, of this type of action being taken on a European level?

Thompson: I don't know of anything that is quite parallel to this. However

there is much pressure on ourselves to move into this area. With, for example, magnetic tape testing, we found that when we asked several centres in Europe to do the tests, they refused.

Cichelli: I think it would be beneficial to have BSI providing a validation service, and a certificate which could be used in publications about compilers.

Wichmann: What impact would this have in the US?

Cichelli: I think it would have a very positive impact, and I know there is a great deal of respect for the careful work done in the context of BSI.

Question: Couldn't the supplier change the tests?

Cichelli: Some vendors do modify the test suite text before performing a validation, but these are getting fewer.

Wichmann: Media conversion is not so much of a problem now that there is an option for square bracket. This character is reserved for national use in the ISO code and corresponds to non-Latin characters in Scandinavia.

Audience: During a validation you must prevent any modification of the source text (excluding permitted lexical substitutions).

Wichmann: The 96-bit parity check program overcomes this problem; running the program is slightly cheaper than compiling.

Morgan: An integral part of the validation service is a document which explains precisely how the validation will be performed. You must obey the same procedures when dealing with things on site.

Wichmann: If for example a hardware error occurs you wouldn't require the validation to start from scratch.

Charter: Vendors must know beforehand what is expected of them, what facilities they must provide etc.

Morgan: The FCTC documents state that the user must provide the validators with reasonable access to the machine and must also provide office space.

Question: What worries me is whether or not the report one would get in America would be the same as one would get here, for the same system.

Wichmann: Anyone can purchase the test suite. But we are discussing third party validation, and I don't know who will perform it in the States.

Audience: We want to sell our product there.

Thompson: Hopefully the various countries will organise themselves so that there is no duplication. I think you must set it up as a world-wide service, give a name to the service and register it round the world. Where you do the testing then becomes irrelevant as long as each centre must be registered in order to be accepted. Once in service, these reports can then

come out in a uniform manner.

Wichmann: But that means agreement must be reached between all the countries before even the UK service can start.

Thompson: Provided the framework is correct at the outset, you can start more slowly.

Charter: I don't know how a UK validation can be accepted in the States. Who would be the acceptance body in the US?

Cichelli: The approach could just rely upon vendors and users reports as we have now. The advantage of a BSI report would be that of an unbiased arbiter of the result. I am not sure that would not be sufficient for the US. Most US companies are likely to be happy with a BSI report.

Thompson: The crux of the thing is an open door policy.

Wichmann: This is true of Ada. DoD have specifically stated that they will encourage a satellite validation facility in Europe.

Addyman: I think if the testing is done by an organisation which has international respect, say BSI, then users from other countries will be impressed even though the testing hasn't been done in their own country. I've seen adverts in England for GSA approved Cobol compilers and that impresses me.

Morgan: If there were a court case in this country over something validated in America by GSA to an ANSI standard, what is the legal position? We don't know the answer. It may be that if a validation is performed somewhere, it would still have to be approved in America.

Thompson: Again, you must resolve this problem of recognition. In the area of TTY compatibility a service has been set up on an experimental basis in ISPRA in Italy. We are thinking of making this service support our procurement and making it available to others. We can publish this in the official journal for the European community. As an official document, it is translated into the Community languages and could possibly be a means of giving European status to the work.

Cichelli: I think that a conformance statement (required by the Standard) plus a BSI-generated validation report appearing on the front of the documentation for a compiler will be very important to a significant number of large companies in the US.

Wichmann: Many Pascal compilers, especially on mainframes, are university produced products and are often distributed without cost. What will happen with these implementations? Will these type of compilers get validated?

Sale: I think they are a dying breed. Pascal grew up in a university environment for five years and escaped from it at least five years ago. A whole host of these compilers are now obsolete and hardly anybody uses them.

Question: University products don't have long term support, so is there any

point in validating them?

Wichmann: University products like the Belfast compiler for the 1900 machines will be around for a few years yet and I don't imagine that ICL will take it up as an official product. University products are important because thousands of students use those compilers and regard Pascal in terms of those compilers. I'd regard them as being prime targets for validation.

Cichelli: I disagree to some extent with Arthur Sale. With major main-frame vendors the suppliers still tend to be from a university environment.

Wichmann: I think I must bring this discussion to an end now. Thank you very much for coming to NPL. I hope you found it useful. Please report any difficulties you may have with the suite, because the quality of the validation suite depends to a large extent on your effort. We are here to provide the suite and eventually hope to have a BSI supported validation service. We can only meet the requirements of the market place if we are told what you think, so please write in.

13

Compiler validation - a survey

R. S. Scowen and Z. J. Ciechanowicz

1. Summary and Conclusions*

There are many problems in defining and standardizing programming languages. This report lists them giving examples from current languages. The difficulties of testing the compliance of compilers to these standards are also analysed. There is then a discussion of alternative strategies for defining programming languages, i.e. by specification languages, formal proof techniques or a standard compiler.

The report surveys ten existing validation suites for nine different high-level languages identifying their strengths and weaknesses. The report also describes the formal validation procedures adopted by the Ministry of Defence for Coral 66 and by the US government for Fortran and Cobol.

The conclusions from this survey are:

1. A validation suite is an important tool, especially if the language standard permits one to check that invalid programs are rejected by a compiler. However testing is only valid for particular versions of a compiler and usually excludes performance measurements and judgements of quality such as the accuracy of diagnostics.

2. Current programming language standards are too imprecise, especially concerning invalid programs. Some languages also make no attempt at achieving program portability which is an important advantage for much software.

3. The MoD system of *accreditation* has advantages over validation in that a manufacturer's own system of quality control is inspected and, once passed, would apply to all software being produced there. However the need for adequate testing and a comprehensive validation suite is not reduced; there are also difficulties applying accreditation when a compiler has been imported.

---

* *Editorial note: This Chapter was orginally published as an NPL Report CSU 8/81 in December 1980. It is reproduced here to make it available to a wider public.*

2. Introduction

The scale of engineering has steadily increased: for example aircraft, power stations and bridges have all become more technologically advanced, complex and expensive. With any large complex system there is a problem of verifying that the finished system satisfies the design specification: how can we tell that it is working correctly? Are tests performed by an independent body or by the manufacturer himself? Does the specification include the tests that will be made? Does the manufacturer or the tester provide any form of guarantee? If the system is replicated, is each example tested to the same extent? Who pays for the verification, the customer directly, the manufacturer (and thus indirectly, the customer) or society at large (a 'free' service)? Very often the tests will have two aspects, firstly, safety (could it fail catastrophically), secondly, performance (does it work well).

Another recent trend has been for engineering and commercial systems to depend increasingly and critically on computers, both in the initial design and later in use. It is thus important that the computers and their programs are reliable and correct. Computer hardware has improved greatly, it is smaller, cheaper, more powerful and more reliable. Computer software has lagged behind: programs are bigger, continually being altered and extended, and often understood by no single person. Without a software revolution, it seems inevitable that there will be catastrophic accidents due to program errors.

A fundamental part of almost all software is the compiler used to translate the program from a high-level language to the computer's own language. Compilers must therefore handle users' programs correctly.

Software is labour intensive and increasingly expensive compared with other computing costs: there is therefore considerable incentive to use programs again, perhaps on a computer with a different architecture. In practice this is only possible if the program is written in a standard language which can be compiled on both machines. Compilers must therefore be compatible with the standard definition of a language: they must also inform the user when his program is non-standard.

This report examines the problems of testing compilers to ensure that they are satisfactory, i.e. they are both correct and compatible with the standard. Unfortunately there are many problems:

Practical considerations ensure that almost all current compiler validation packages treat the compiler as a black box, i.e. look only at the output from various test cases. However it is well known that although this method may detect some errors it is quite incapable of proving the absence of any other errors.

At the current state of the art almost every large computer program contains errors. It follows that a compiler, itself usually a large program, will contain errors.

Compilers, like other computer systems, are subject to continuous modification. Must a compiler be retested after each modification? A change to the operating system can also wreck a compiler: this increases the need for retesting.

Each individual part of a compiler might be satisfactory on its own, but a fault might be caused by two parts interacting in an unforeseen way. It is usually impossible to test all cases.

There is a similar problem in that compilers commonly provide a whole range of optional facilities, for example listings, cross-reference indexes, runtime error checks, different character sets, independent compilation. The options used when testing the compiler may be satisfactory when other untested cases would fail.

The pressure of the market place encourages computer manufacturers to *improve* their compilers by adding extra facilities. But it is impossible to test arbitrary extensions to a language.

Existing standards for programming languages are written in more or less formal English. Inevitably this process has resulted in standards that are sometimes vague, ambiguous and incomplete.

It is sometimes difficult to know whether results from tests are satisfactory or not, for example the measured accuracy of floating-point arithmetic.

The quality of a compiler cannot be judged only from its ability to compile and execute a few correct programs. The maximum size of a program may be too small, programs may run too slowly, the time for compilation may be too long, there may be insufficient help to debug incorrect programs.

There may be anomalous cases which are practically impossible to detect with random tests.

Compiler writers will run the test suite as a final check that a new compiler is working. Any errors shown will be removed and only then will it be submitted for formal validation. Naturally the compiler will pass first time and users will perhaps have a misplaced confidence in the apparently error-free compiler. They will fail to realize that formal validation will not have reduced the likelihood of other sorts of errors and problems.

Optimizing compilers provide special problems both in testing that optimization has been performed correctly and that language features are always compiled correctly.

All these problems are considered in further detail below.

Of course all software should be correct and efficient and there are probably other software packages that should be validated, e.g. libraries of numerical software. However the methods may be rather different and the possibility is not considered further in this report.

Certification or validation

When all the technical problems have been solved, it must be decided whether compilers are to be certified or validated. These two terms are sometimes used interchangeably, but have different meanings. If a compiler is to be certified correct, this is equivalent to a guarantee that there are no faults and the certifying organization might need to defend a legal action

in the event of any defects being discovered later.

Validation is considerably weaker, it merely implies that the compiler has been submitted to a particular set of tests on a particular occasion. The innocent user may be shown an impressive certificate recording this event; the more experienced will judge for himself the worth of the validation. For example, BSI are liable for negligence in performing a specific test, but not liable because a test is inadequate and a tested product fails.

There is no doubt that certification is impossible with current languages, their standards and compilers. Equally validation is only as valuable as the test suite.

A comparison with validation and certification elsewhere

Of course compilers are not the first complex products that need to be tested for conformance to a standard and for adequate performance. Before deciding on a method for judging compilers, it is sensible to see what can be learnt from the methods and experience of other industries.

Section 7.6 describes very briefly the process for granting a certificate of airworthiness for a civil aircraft. The comparison with conventional compiler validation is marked; the Civil Aviation Authority is involved during the design and manufacturing stages, and not just when the aircraft is finally complete. Certification is also a continuing process, faults discovered subsequently must be reported and repaired.

Accreditation

The Ministry of Defence pursue a more indirect method of ensuring that goods are of satisfactory quality. They use a method called quality assurance which does not test the product directly, but rather confirms that a firm is competent and able to produce satisfactory products. All details of a firm will be considered; as well as looking at the methods of production and internal quality control, the financial structure, personnel, facilities, and method of project management will all be examined.

Accreditation is known formally in MoD as the assessment of contractors to Defence Standard 05-21. Assessments are conducted by specialist members of MoD from Research and Development, Project Management, and Quality Directorates dependent upon the company's products. MoD are now cooperating with some of the nationalized industries to form joint assessment teams thereby reducing overall costs as many nationalized industries are committed to the same form of accreditation as MoD.

The process starts with a preliminary plan and visit. Often all branches of a firm are examined but this depends on the firm's request and MoD's requirements. The examination is wide-ranging and a brief description is given in Section 7.7; it results in a form reporting the findings and outlining the deficiencies or recommending acceptance. When the firm is judged satisfactory it is included in MoD's *List of Assessed Contractors* (LAC). Reassessment is normal (at 2 or 3 year intervals), but actually "whenever thought necessary", for example if customers report faults, or major changes

in personnel are known. Spot checks are also made.

The cost naturally varies with the size of the firm, but is typically a few man weeks or months.

Since 1976 MoD have required contractors involved with software design to meet Defence Standard 05-21, for example by October 1980 nine software houses were assessed satisfactory. A similar scheme could be adopted for compiler validation. Instead of an independent tester running a validation suite to certify the compilers for a particular language, he would ensure that the compiler writers use the validation suite as part of their quality control process. Note that accreditation can be extended to cover, not just the compilers of a single programming language, but any other software for which a comprehensive validation suite exists.

There is one problem with accreditation (Contractor Assessment): although it shows that the company is capable of controlling and developing projects to Defence Standard 05-21, it rests with the customer for each individual project to specify the Defence Standard as a contractual requirement.

Impact on standardization activities

The standard for a programming language is not just the result of academic exercises. The objective is usually an attempt to regularize the de facto definition of a programming language for which there are many different compilers on many different computers. Naturally each manufacturer has a large vested interest in maintaining the validity of his own software and fights strongly to ensure that his compiler meets the standard. The consequence is that standards are defined permissively and with all the faults mentioned elsewhere in this survey.

More effort on compiler validation will be largely wasted unless as standards are updated, they are made more complete and precise.

3. Problem areas

Experience has shown that almost every large computer program contains errors. A compiler is itself usually a large program and can be expected to contain errors.

This chapter considers in detail some of the problems in validating compilers.

Testing a black box

Almost all current compiler validation packages treat the compiler as a black box, i.e. look only at the output from various test cases. It is well known that this method may detect some errors but is quite incapable of proving the absence of any other errors. Unfortunately this is the only economically viable method currently available: the source text is often a commercial secret and unavailable, and even if it were available there are few suitable techniques for analysing its properties. Further, validation suites can be based on past experience, e.g. test for the existence of faults that have previously

occurred, and use the knowledge of how compilers are constructed. Validation suites can also be continually extended so that although there can be no certainty of detecting all errors, we can at least ensure that they do not recur.

The need for retesting

It would be a serious fault with any compiler validation system if compilers were never retested. As fresh releases of the compiler are made and the tests altered, the original results can no longer be regarded as valid. Retesting is thus desirable and probably essential; the knowledge that an earlier version of software was valid has little value to a user: all programmers learn rapidly from experience that changing a program can introduce errors in seemingly unrelated places. It would be helpful to users if each compiler listing were required to state which version was last validated and when it took place.

The only way to be certain that compilers are still working is to insist on complete periodic retesting. Sometimes this must be done by an independent contractor; however it may be possible to license the compiler writer to perform the validation himself.

Language extensions

The US Federal Government has defined four levels of Federal Standard COBOL which form a sequence of nested subsets of the full ANSI Cobol Standard. As part of the validation requirements, each compiler must be able to flag on a compilation listing all uses of language features which are not included in a specified level. The implementer must provide a compile-time switch by which the monitoring can be set to any level at or below the Federal level which the compiler claims to support. The facility is validated by compiling a program (in practice several programs) four times, once for each position of the switch. On the listing, all features from above the level at which the switch is set should be flagged and nothing from below that level should be flagged. No attempt is made to test the treatment of extensions beyond the standard.

This is a partial solution to a severe problem: it is very rare for a compiler to be written only for the Standard version of a language. Manufacturers almost always include extensions in order to increase the attractiveness of their computer systems. The extensions often make life easier for the programmer but are a hindrance to program portability. Validation suites usually completely ignore language extensions. This is a serious deficiency: most users are not aware which features in a given language are standard and which are extensions, they would be very surprised to learn that even when the compiler has been certified correct, the guarantee may not apply to many parts of their programs.

It is clearly impossible for a validation package to test extensions to the implemented language; an independent validation organization cannot even provide a list of the extensions because there is no way of knowing that the programmers' manual is complete. Besides the extensions openly described in the manual, there may be hidden extensions known to the implementer but not publicized, and accidental extensions unknown to anyone and waiting

to be discovered by accident.

Thus it is an important requirement that the compiler is capable of warning a programmer whenever he contravenes the standard language. This is more easily said than done. Some extensions can easily be detected during compilation, for example the addition of **complex** type to Algol 60.

Other extensions can only be detected at runtime, for example suppose the Fortran 77 language has been extended so that:

```
REAL AA( 1 : 0 )
```

is not treated as an error but as an empty array, no errors will be reported unless the program tries to access an element of the array. For this extension the compiler would have to check adjustable array declarations (i.e. those in subroutines that have variables specifying the bounds) at runtime and print a warning message when necessary. Because the array declaration might be executed many times, it would be an essential refinement that each sort of warning message is printed at most once for each run of the program.

Unfortunately, some sorts of extensions are practically impossible to detect. Consider the Algol 60 statement:

```
xx := zz + ff(xx);
```

where ff is a function that has the side effect of assigning a value to zz. The statement is ambiguous and therefore illegal because Algol 60 does not define the order of evaluation for the two primaries zz and ff(xx). This sort of error cannot be detected easily at translation time or runtime; it is necessary to perform a data-flow analysis (as in DAVE [OSTE76]). In even more complicated cases (e.g. replace zz by elements of an array where the subscripts sometimes have the same value), no formal process may be able to detect the error at all, only warn that the program is possibly illegal.

The compiler validation scheme should probably require the compiler writer to provide suitable tests for each language extension (or use previously written tests where the same extension has been implemented earlier). The extra programs should demonstrate that the extensions have been implemented correctly and that they interact correctly with standard features of the language and with other extensions. When compilers are retested, successful execution of the extra tests would give reassurance that the extensions had not been altered.

With Ada, a different approach is being adopted: any organization submitting a compiler for validation must certify that no extensions to the Ada language are known to be present.

The vagueness of current standards

Programming language standards are not usually a model of clarity. Some of the difficulties are deliberate because that was the only way the standards committee could agree on a definition. Precision would have caused some manufacturers' compilers to be non-standard, and enormous costs to be incurred by altering compilers and programs to comply with the standard.

More often the ambiguity arises because programming language standards are usually written by computer experts who have had little previous training in drafting standards. This has often led to ambiguity and incompleteness.

It is all too easy to make assumptions that are nowhere specified explicitly.

For example, current language standards are often delightfully vague, even about fundamental aspects of a language, sometimes they say practically nothing. Thus, the only information to be found in Fortran 77 [ANSI78] concerning real (usually but not necessarily floating-point) arithmetic is the following brief remarks:

"Evaluation of an arithmetic expression produces a numeric value" (page 6-1, lines 11-12);

"An integer datum is always an exact representation of an integer value" (page 4-3, lines 3-4);

"a real datum is a processor approximation to the value of a real number" (page 4-3, lines 14-15);

"This standard does not specify: ... (6) The range or precision of numeric quantities and the method of rounding of numeric results", (page 1-1, lines 33, 53-54).

There is no requirement for the result of a multiply operation ever to be even approximately correct. Thus the standard is so nebulous that it would probably be impossible to fail a compiler because the arithmetic operations were insufficiently accurate. At present this has not mattered because there is no agreed way of measuring the accuracy of computer arithmetic. This problem is considered in more detail in the next section.

How accurate are arithmetic operations and functions?

Consider the program:

```
      REAL  XX, ACC
      ACC = 0.7 + 0.3
      ACC = ACC - 1.0
      WRITE(6, 990) ACC
      STOP
  990 FORMAT(32H ERROR IN COMPUTING 0.7 + 0.3 = , E15.4)
      END
```

What would we learn from such a program? The values 0.7 and 0.3 cannot be represented precisely in a binary computer, the addition operator might truncate or round the result, the WRITE statement might even truncate values smaller than a certain (non-zero) magnitude to zero. There are so many unknown features in the program that no useful deductions can be made from the results.

Several attempts have been made to overcome these problems:

Cody and Waite [CODY80]
have developed a carefully written suite of programs to investigate standard functions. They compute the value in two ways and report on the differences. The results are thus objective but measuring neither the absolute or relative accuracy of the standard function. Their suite is used

in the Pascal validation suite.

Bailey and Jones [BAIL77]
have performed extensive tests on the CDC 6600/7600 single-length library functions by comparing the computed values with the values obtained by computing the corresponding double-length function with the same arguments. This makes the reasonable assumption that the double-length function is completely accurate over single-length. One disadvantage with this process is that assumption is not infallible, [BROW80] quotes a case where the comparison of double-length real values considered only the most significant parts of the values. A more important disadvantage is that there is no way of extending this technique to test the double precision functions.

Dekker [DEKK71]
showed how to simulate double precision arithmetic using single-length operations. However it is necessary for the single-length operations to be 'optimal' or 'faithful'. He points out that such properties can easily be guaranteed and are possessed by some computers. Unfortunately they are not possessed by all computer systems and so this technique cannot be used generally.

Another strategy is to compare the computed result with the correctly rounded result read from a data file. This would require a routine to read a value expressed in binary notation. The idea is examined further in Section 7.1.

These techniques are not entirely straightforward. They usually require knowledge of the floating-point representation in the actual hardware; some of the required properties can be found using Malcolm's algorithm [MALC72].

Sometimes calculations will give a result that represents an 'undefined', 'infinity' or un-normalized floating-point value. The meaning of such a value is not prescribed in many standard programming languages (e.g. Fortran) and it will be desirable to perform no further computations with it.

The floating-point and arithmetical properties of computers are now being considered with the aim of defining standard properties. An issue of SIGNUM Newsletter [SIGN79] discusses the need for a standard and outlines three candidates for a standard. Some other recent work considering the problems and intricacies of floating-point arithmetic includes Brown [BROW79] who develops a model of floating-point arithmetic which can describe the properties possessed by most computers, the Ada rationale [ADA 79] which explains the reasoning behind the fixed and floating-point facilities provided in Ada, and Reinsch [REIN79] who describes what he and other numerical analysts like to see in floating-point arithmetic.

The quality of a compiler

There are many important aspects which contribute to the value of a compiler and which are likely to be left untested by a validation suite because they are machine-dependent. In particular:

The compiler should be well integrated with the operating system so that for

example, source programs, data, and results can all be edited, filed and printed. The compiler should be usable in the same way as compilers for other languages with a simple, natural job-control language that is nevertheless comprehensive.

The compiler should give simple clear messages when the program contains a syntax, semantic or runtime error. The error messages should refer to the program in the language used by the programmer, and not specify machine addresses or instructions. Better still the compiler should also warn the programmer when the program is odd. Section 7.5 (The value of warning messages) shows the need for this.

A compiler should be efficient with correct programs, as well as helpful with wrong ones.

If the compiler can be used in a multi-access environment, programs can be tested and developed more efficiently than if they must be prepared and submitted as a deck of cards.

Program development is also much more difficult if the turn-round time for an edit, compile and run cycle is a day rather than a few minutes. Even if the speed is adequate on one particular model, sometimes the compiler can be slower by anything up to 100 times if there is too little main store and/or backing store.

Even if a compiler is otherwise perfect, it will be most inconvenient if programs fail to compile or run because they are too big, or because the data structures are too big.

A compiler must be robust; neither it nor its compiled programs should ever lose control and end by giving a hexadecimal or octal dump.

It is important that the computational costs of a program are not excessive. But remember costs occur not just from compiling and executing a program, they also arise from preparing, storing, loading, and amending it.

There should be good libraries of procedures available for performing common tasks. And it should be possible for programs to call them even when they are written in another language.

Anomalous cases

Pascal specifies that integer arithmetic is guaranteed correct if the operands and result all lie in the the closed range [-maxint, maxint] where 'maxint' is some implementation-defined value. How can this requirement be tested exhaustively, for example Burroughs B6700 has:

```
maxint = 549 755 813 887;
```

If one positive and negative value can be tested every microsecond it will require some ten days of computer time merely to test each integer is allowed. What if maxint is a thousand times bigger? Unfortunately this is only one of many similar problems and in each there may be an anomalous case practically impossible to detect with random tests. Section 7.2 shows an

example from a language implemented at NPL.

Bailey and Jones [BAIL77] also discuss this problem in the summary of their report when measuring the accuracy of standard functions: "the prudent reader may well assume that the actual errors are at least as large as we have observed".

There is a related problem in that a compiler is not just a single program but more a family of related modules providing a range of optional facilities, for example listings, cross-reference indexes, runtime error checks, different character sets, independent compilation, and syntax-check or compile-only or compile-and-run. The options used to test the compiler may be satisfactory when another combination of options would fail.

Tuning the compiler to the tests

Compiler writers will naturally want the tests to be public so that they know the performance necessary to pass, and know the interpretation to make in ambiguous areas of the standard definition of the language. This requirement makes it necessary to publish details of the test suite and the results required for a compiler to pass. In any case, it would probably be impractical to keep the validation suite secret because it would be run on so many different machines at so many different sites. Yet by publishing the test suite, there will obviously be a temptation to tune the compiler to pass the validation tests and make little or no attempt to discover and cure other faults. This strategem can be partially foiled by parameterizing some tests; however completely random tests would make it impossible to confirm that an altered compiler gives unaltered results where appropriate.

Thus the validation suite must be comprehensive and programmers must realize how much (or little) is implied by any claim that a particular compiler has passed the required validation tests. Inevitably a requirement for comprehensiveness increases the cost of both writing the test suite and carrying out the tests.

Optimizing compilers

Optimizing compilers try to recognize the parts of programs that can be compiled in shorter or faster code. Compilations take longer but the result is better. Unfortunately testing such compilers poses extra problems.

Some test programs may be so simple that the compiler optimizes away the feature being tested. For example, by performing as much at compile time as possible, an optimizing Algol compiler may transform:

```
begin
    integer  ii, jj;
    ii := 3;
    jj := 2;
    if ii > jj then
        out string(' Greater_than_for_integers_OK`)
    else
        out string(' Greater_than_for_integers_fails`)
end
```

into the trivial:

```
begin
        out string(' Greater_than_for_integers_OK`)
end
```

This will tell us that '>' is optimized correctly at compile-time, but nothing about the runtime behaviour. The problem is serious because the printed output from the test suite cannot indicate the failure to make the required test.

On the other hand, it may happen that the programs in the test suite are so complex that the compiler doesn't attempt any optimization; if so, the optimizing facilities are virtually untested.

For some languages, e.g. Ada or Fortran, independent compilation of two subroutines may ensure a satisfactory test. Unfortunately there may be an even more intractable problem. It may happen that the optimization is perfect for simple cases, and also perfect for very complicated cases where the compiler recognizes its inability to perform optimization. However at the boundary, some cases are compiled wrongly. Unless the code of the compiler or its output is examined, only luck will discover these faults.

4. Alternative strategies

It has been assumed that compilers should be judged in the future as they have been in the past, i.e. by constructing a test suite of programs which are submitted to a compiler, and examining the results to see if the compiler passes or fails. It may be that there are alternative strategies which could give the desired objective of reliable software more cheaply and easily. This section examines some of the possibilities.

A standard compiler

An old idea worthy of reconsideration is to regard the compiler as the definition of a programming language. This idea was first suggested by Garwick [GARW66]; it was then generally regarded as impractical, compilers were mostly written in low-level languages and almost incomprehensible; there were also very many completely different computer architectures in use. However, since that original suggestion, languages such as BCPL, Pascal and RTL/2 have been designed and implemented. In each case, the first compiler was written in the language itself and almost all subsequent compilers based on it to a greater or lesser extent.

This approach, besides ensuring portability and a method of implementing the language cheaply and quickly, ensures that there are no gaps in the language definition. A standard compiler may even be more compact than a standard written in English; it certainly need not be more difficult to understand.

Specification languages

A programming language standard can be regarded as the specification of a program, i.e. a compiler for the language. Recent work on formalizing the specification of computer systems could have several benefits, for example a formal specification is essential for any attempt to prove that the corresponding suite of programs is correct. Formal specifications would also make possible processors to check that the specification is:

Complete: all possibilities are specified.

Consistent: different parts of the specification do not contradict each other.

Not redundant: no requirement is stated more than once.

Unambiguous: so that there are no misunderstandings between the customer and implementer.

Not over-specified: so that the requirements do not unnecessarily constrain the implementer.

A specification language normally provides a functional definition rather than an algorithmic definition of the system and bridges the customer, the implementer and their management. English is unsuitable for a specification language. It is too easy to be ambiguous and for large systems it is shown in [JONE79] that English is far more prolix than the code required to implement the system.

A recent paper [DAVI79] describes various ideas used and suggested for specification languages, for example:

Finite state machines define the outputs resulting from each input but become complicated when it is necessary to define synchronization.

Stimulus response sequences use a pseudo-Algol definition of the responses from user stimuli.

Petri nets (see [PETR62], [PETE77], [BRAU80]) are a graphical technique that express synchronization requirements extremely well. However their graphical nature makes them more useful as documentation output from the requirements phase rather than as an input specification.

Perhaps one day specification languages will show the way to better standards for programming languages. But current methods are inadequate for today's complex programming languages.

Program proof and formalized testing

An idea that is attractive at first sight is to prove a compiler is correct; in fact some research has been done on proving programs are correct in the same way that geometrical theorems are proved in Euclid. However the results, as far as compiler validation are concerned, are not encouraging. It is essential to have a specification and to have the text of the program available: neither requirement can be satisfied with normal commercial compilers. Although researchers (e.g. D Bjorner on Chill and Ada, Belz (TRW) on Ada in Semanol) are developing operational formal definitions (for an introduction to these ideas see [BJOR80]), a formal specification is not generally available and the text of a compiler is often a closely guarded trade secret. The task of finding techniques to analyse arbitrary programs is recognized as intractable, for example, it is not even generally solvable whether or not a program goes into an infinite loop. Strachey showed [STRA65] that there is no program which can read the text of an arbitrary program and report whether or not it terminates. All successful research in this field has been done by constructing a program to perform a specified task and then proving that it does so. But is the proof correct? An example pointing out an incorrect 'proof' is given in [GOOD75].

The theory of program testing is equally discouraging. Weyuker and Ostrand [WEYU80] point out that for every element d in the input domain of a program, there is a program which processes every element other than d correctly, but is incorrect on d. This implies that correctness can only be guaranteed by testing every possible case. Thus test cases that execute every part of the program, or even execute every program path, cannot guarantee correctness either. For an example, consider the following program which is based on an example by Goodenough and Gerhart [GOOD75]:

```
begin
    integer  ii, jj, kk;
    read integer(ii, jj, kk);
    if (ii + jj + kk) = 3 * ii then
        out string(' The_three_values_are_equal`)
    else
        out string(' The_three_values_are_unequal`)
end
```

A more practical aim is to demonstrate that particular errors are absent. With this technique, assume a particular error has been made and construct test data which will disclose it. Foster [FOST80] gives a method of systematically generating test data to do this.

5. Existing validation suites

A number of schemes already exist for validating compilers. The Ministry of Defence tests Coral 66 in this country, and there is increasing validation of Cobol and Fortran compilers in USA. This section briefly describes some of these validation suites and how they are used.

Ada

Ada is a new language sponsored by the US Department of Defense (DoD) which is designed to be "a language with a considerable expressive power covering a wide application domain". The aim is that Ada should be a "common" language throughout DoD, i.e. used in all computer systems. It is intended that any compiler used on a DoD project must have been formally validated. The first reference manual was published in April 1979 with a revised edition a year later. Many related projects are being supported including the provision of an Ada Compiler Validation Capability (ACVC). This work is being carried out by SofTech; it is described in a long range plan [SOFT80a] and a paper [GOOD80].

Three phases are planned:

(1) October 1979 – October 1980: To produce a baseline set of tests and the tools needed to run them. Some of the features not covered thoroughly include tasking, the input/output package, and optimizations.

(2) October 1980 – September 1981: To extend the baseline test set and to validate the test suite itself by using it to validate a compiler.

(3) October 1981 – April 1983: To extend and maintain the test set further, and to develop new approaches that attack problems not fully solved earlier.

The long range plan recognizes that a validation suite cannot by itself show all the issues of interest to potential users, although these properties ought also to be gathered eventually by the ACVC.

The tests will be designed by presupposing the sorts of errors that implementers may make and then constructing tests that can only succeed if the error is absent. However there will be no attempt to discover errors that are thought to have no practical importance.

The test suite of programs is only part of the ACVC programme. SofTech will also write an "implementers' guide" and prepare "validation support tools" to assist in running the tests and analysing the results.

The implementers' guide describes the ramifications of the Ada Report, especially aspects that must be implemented in a particular way. It also specifies details of the various validation tests to be written. The guide will be structured by considering each subsection of the Ada report in turn. The first draft [SOFT80b] discussed the most stable parts of Ada, a final version [SOFT80c] has been published in October 1980.

The test suite will be written as many small test modules, each testing one particular feature so that the suite identifies the language features that are correct as well as those that are faulty. The chief disadvantage of so many test modules (about 1000 are forecast) is the large cost of compiling and running so many programs. It is therefore desirable to provide support tools to ease the burden, and facilities to group the tests into larger units when appropriate. The validation tests will be divided into six classes according to the expected result. The suite will test incorrect as well as correct programs, and check both syntactic and semantic errors. Some tests will indicate approximately the capacities supported by the compiler (for

example the maximum number of array dimensions). The standard libraries, for example input/output, will also be tested.

One relatively unusual feature of Ada is that programs will frequently be compiled on one machine (the compiler host) and executed on another (the object host). SofTech also envisage a third machine (the validation host) which will be used to prepare the test suite and analyse the results. This is because extensive editing and file processing will be necessary to run the tests, in particular the tests will need to be prepared so that they are consistent with the machine dependent features of the compiler and execution hosts.

An Ada Compiler Validation Organization (ACVO) will organize and perform validations of Ada compilers in a similar way to the ones performed by the Federal Compiler Testing Center for Cobol and Fortran. A report will be published for each compiler validated and revalidation will be required within two years.

ACVO are well aware of the problems of compiler validation described in an earlier section and are trying to overcome them, for example any organization submitting a compiler for validation must certify that no extensions to the Ada language are known to be present.

After successful validation they will require copies of the source and object codes for the compiler. These will be archived and used to resolve any claims that a compiler has been changed and to protect the US Department of Defense against bankruptcies and other unforeseen circumstances.

Optimizing compilers will be tested with all optimizing options enabled, but ACVO may repeat the validation exercising different options.

ACVO intend to collect performance data during formal validations and publish the results in an appendix to each final validation report. ACVO will also collect reports of errors undetected during formal validation and develop tests to ensure their detection in future validations.

There will be a charge for validating an Ada compiler: a base charge estimated at $5000 to cover the cost of maintaining the ACVC and an additional charge to cover all incidental expenses.

Algol 60

A set of 130 test programs was produced for Algol 60 at NPL during 1973 [WICH73]. These were extended to 180 tests in 1975. Most of the tests are very short and aim at testing one feature of the language. The testing is aimed at the more difficult aspects of the language, since these areas are more likely to contain errors. The initial effort was about 9 man-months. Many of the tests have since been converted into Pascal and are in the existing test suite [WICH79]. The results of running the test suite on six compilers are reported in [WICH76] and illustrate that erroneous programs are twice as likely to indicate compiler bugs as correct programs. The sixth compiler tested was special in that the compiler writer used the tests to debug the compiler and in consequence, got nearly bug-free results. The test suite has not been updated to conform with the latest "standard" for Algol 60 [DEMO76].

nor is any further work on the test suite envisaged.

Algol 68

This is not a formal test suite but a collection of programs which can be used to test an Algol 68 compiler. The collection was not produced as a single suite but has grown over several years. Some of the first programs were written as desk exercises by staff at the Mathematisch Centrum as they studied Algol 68 and learned of its power. The initial spur to their collection came when an interpreter and compiler became available and it was necessary to find some programs to test them. Naturally the existence of compilers encouraged further test programs to be written that (some hoped) might cause the compiler to fail.

The suite was used extensively when testing the Control Data Algol 68 compiler; it was first published in 1976, and a revised version in 1979 [GRUN79]. Since then copies have been distributed widely and further modifications made. In April 1979, IFIP Working Group 2.1 on Algol accepted the formal resolution:

> "This Algol68 Test Set has been reviewed by IFIP Working Group 2.1, which considers it as a valuable means of testing implementations of Algol68."

Although the test set has grown rather than being systematically designed, Grune states:

> "In my opinion, if a compiler processes the test set well and works well on the daily stream of average programs, it is a very good compiler. Through its unusual complexity, the test set will uncover most incorrect short-cuts, and the constant use of simple features will prevent the compiler from being too much tuned to the test set. I may have to make an exception for a heavily optimizing compiler. The present test set may be less effective there since such a compiler would often decide not to do any optimization at all: a special test set is needed based on knowledge of the optimization techniques."

The test suite is listed fully in [GRUN79] and can be obtained in various machine-readable forms from the Mathematisch Centrum in Amsterdam.

Basic

The original version of the language Basic was developed at Dartmouth College, USA, in 1963-64 by Professors Kemeny and Kurtz. Subsequent versions of Basic extended the original definition by including various additional features. Starting in 1972, work took place in ANSI X3J2 and in ECMA TC21 to define a standard which would be a language core satisfied by as many implementations as possible. ANSI and ECMA prepared two technically compatible standards, the ISO draft standard [ISO 79] being based upon them.

A test suite based on the proposed ANSI Standard [ANSI77] was consequently designed by D E Gilsinn and C L Sheppard at NBS to test the conformance of Basic processors with the Standard. The test suite is described fully in [GILS78]. The general objectives of the suite were to examine a processor's ability to:

(1) handle syntactically correct programs

(2) interpret semantics correctly

(3) return required exception condition notifications.

Other people also contributed towards the tests, including members of the ANSI subcommittee X3J2 and the designers of the NBS Fortran suite [HOLB74]. Version 1 of the test suite was published in January 1978, having needed three man-years of effort. The NBS Basic suite will eventually be used for Government-wide validation of Basic processors procured by Federal agencies. Future releases of the suite will depend on the availability of more comprehensive tests and on any changes made to the standard. Certain extensions to the language will eventually be made in such areas as files, flexible I/O formatting, mathematical functions and matrices, control of real-time processing and string manipulation.

The majority of the tests are relatively short and the largest programs contain not more than 300 lines. The programs are self-contained and are ordered by increasing levels of difficulty. Also later programs in the suite rely only on facilities which have already been tested. Thus the simplest facilities like 'print' and assignment of values (both numeric and string) to variables are tested first.

The output from the validation suite requires careful scrutiny due to the format of the output statements used in the suite. A typical piece of output is:

IF THE NUMBER PRINTED AFTER THIS STATEMENT IS NEGATIVE
AND THE MACHINE INFINITY FOR THIS SYSTEM, THEN THE TEST
WILL HAVE PASSED.

Version 1 contained 161 programs, each examining an isolated area of the standard (each file contains one executable program). In March 1979 it was decided to update the test suite, in response to suggestions from users. Version 2, with 208 programs, required an additional 15 man-months, almost all programs being rewritten to some degree.

The new test suite is described in a two-volume report [CUGI80b], the first volume describes the tests and methodology, the second contains a listing of the programs together with sample results. Version 2 has a much improved investigation of errors and exceptions. Cugini writes:

> "The standard for Basic ... attempts to specify what a conforming processor must do when confronted with non-standard circumstances. There are two ways in which this can happen: 1) a program submitted to the processor might not conform to the standard syntactic rules, or 2) the executing program might attempt some operation for which there is no reasonable semantic interpretation, e.g., division by zero, assignment to a subscripted variable outside of the array. In the Basic standard, the first case is called an *error*, and the second an *exception*, and in order to conform, a processor must take certain action upon encountering either sort of anomaly.

A program with an *error* must either be rejected with a suitable message to the user, or the processor must be accompanied by documentation which

describes how the program will be interpreted. When an exception occurs in a program, a processor must first report the exception, and then, according to the type of exception, either terminate the program, or apply a recovery procedure and continue.

These requirements have made it impossible for the test suite to report whether a processor conforms to the standard or not; instead, as Cugini writes:

> "[The test programs] are best seen as one component in a larger system comprising not only the programs, but the documentation of the programs, the documentation of the processor under test, and, not least, a reasonably well-informed user who must actively interpret the results in the context of some broad background knowledge about the programs, the processor, and the language standard."

The only requirements in the standard regarding accuracy in evaluating mathematical expressions are that numbers should be printed in a form that exhibits at least six decimal digits. Version 1 used extended precision routines to test the accuracy of the supplied standard functions. These routines produced numbers having more than six digits accuracy which were then rounded to six digits. These "true" values were then compared with values generated by the test programs. Version 2 rejected this strategy as too vulnerable to the problems of cicularity. Instead the required results are read as data and the tests require only simple IF statements comparing constants and variables. Some tests that attempt to calculate the accuracy of computations are for information only.

Version 2 also includes a number of informative tests that investigate the performance of the RND function that generates a sequence of pseudo-random numbers.

Cobol

(Written by P R Brown, National Computing Centre Ltd)

Cobol is one of the languages for which the United States Federal Government operates a compiler validation service. A recent paper by George Baird [BAIR79], manager of the Federal Compiler Testing Center (FCTC) gives a full description of this service, its history and the nature of its tests. The user's guide to the current test suite is available from the United States National Technical Information Service [NTIS79].

Cobol compiler validation has a long history. Writing test programs was in the program of work established at the first (January 1963) meeting of a committee which evolved into ANSI X3J4, the principal standardization body for Cobol. In 1967 the United States Air Force and Navy began separate work on systems which could be used to measure Cobol compilers against the standard. (Although the first Cobol standard was not formally approved until August 1968, its content was known with reasonable certainty by early 1967). The Air Force project was for a full system in which test programs were to be adapted to different computers by means of parameter cards. The Navy project was more in the nature of a short term feasibility study and began a few months after the USAF work.

Making use of the work already done by the USAF and by ANSI, a set of audit routines was produced by the Navy Programming Languages Group under the direction of Commander (now Captain) Grace M Hopper, USNR. These audit routines had the capability of displaying for analysis actual as well as expected results when tests failed. The first version of the audit routines consisted of twelve programs and a total of about 5000 lines of source code. The tailoring of a program to a particular compiler or operating system was done manually. The Navy project confirmed that Cobol compiler testing was feasible and suggested techniques for the development of audit systems.

By version 4, released in December 1969, the Navy Audit Routines consisted of 55 programs and about 18,000 card images. These were capable of testing the full 1968 Cobol standard and were used both as benchmark and conformance tests in Navy Cobol procurement. In December 1970 the US Navy was given the task of creating and operating a Cobol Compiler Testing Facility for the whole Department of Defense, making use of its own experience with the Audit Routines and that of the US Air Force with its Cobol Validation System.

In 1973 the scope of the Navy compiler validation service was expanded to cover the needs of the whole of the US Federal Government. At about this time it began validating Fortran compilers as well as Cobol compilers. The Federal Compiler Testing Center was transferred to the General Services Administration in May 1979. As part of the GSA, the FCTC continues to carry out validations and to develop the two test suites. The most recent full release of the Cobol test suite, version 3.0 for the 1974 standard, consists of 323 programs and 218,560 lines of source code.

Some programs have been revised and new programs written for version 4.0, which is due to become the official test set in January 1981.

In principle, the FCTC conducts compiler validations for the purposes of the United States Federal Government, but in practice the results of the validations are freely available. Out of almost 105,000 possible combinations of modules from the ANSI Cobol standard, the US government has selected just four as levels of Federal Standard Cobol. Every compiler procured by the US government must conform to one of these levels, must produce object programs capable of accepting and producing files recorded in ASCII, and must provide a facility for the user to specify a level of Federal Standard Cobol against which his program is to be monitored at compile time. The monitoring is an analysis of the syntax used in a source program against the syntax included in the specified level of Federal Standard Cobol. Any syntax used in the source program that does not conform to that allowed by the user-selected level of Federal Standard Cobol is identified in a diagnostic message in the source program listing.

The FCTC issues a Certificate of Validation, which is current for one year. It takes care to point out that this only certifies that the compiler has been submitted to its tests, and not that it provably conforms to the standard. The tests are intended to be comprehensive and in practice appear to exercise most syntactic and semantic areas.

The test suite is maintained by the FCTC itself. It has evolved from the original Navy Audit Routines and is still being extended. Most of the

development is concerned with testing the interaction between language elements, although work will soon start on a major revision to produce a test suite for the revised Cobol standard due in 1981.

Since 1973 the FCTC and its predecessor have carried out just over one hundred Cobol compiler validations. 41 Cobol validations and revalidations are scheduled for 1980. In June 1980 there were 34 companies from 17 implementers on the FCTC Certified Compiler List, but only five of the compilers had passed the tests with no errors. The list of certified compilers is updated each month by the

Federal Compiler Testing Center,
5203 Leesburg Pike, Suite 1100,
Falls Church,
Virginia, 22041,
USA.

Apart from one or two specific areas, functional testing is performed on all syntax and semantic features of the language. Because of the vast number of possible combinations, (about six million combinations of data descriptions for the simplest form of ADD statement), testing for interaction between features is much more selective.

The principal areas not tested are the Communications facility and arithmetic expressions. The Communications facility was a new feature in the 1974 standard and gave rise to a relatively large number of requests for interpretation. Execution of programs using the feature also requires the presence of a Message Control System (MCS) which is defined only in outline in the standard. These problems have now been resolved and tests on the Communications facility will be included in the next release (version 4.0) of the test suite. The syntax of arithmetic expressions and the syntactic environments in which they may be used are well defined in the standard. However, the techniques used in evaluating expressions are left to be defined by the implementer. The United States National Bureau of Standards (NBS) has stated that this means that the results of evaluating arithmetic expressions cannot be predicted and therefore the interpretation of arithmetic expressions cannot be tested. This state of affairs will be changed only by revision of the Cobol standard. In the draft for the 1981 COBOL standard, a floating point representation is prescribed for all intermediate results in arithmetic operations. The features prescribed include the length of the mantissa and the rules for reduction of values to that length. At the time of writing (October 1980) there still seem to be a few minor problems in this area, but they appear capable of resolution.

The programs of the test suite are organized into groups, each of which is aimed principally at one of the twelve modules in which the Cobol standard is presented. The groups are further subdivided to reflect the two levels of most of the modules. All the programs produce reports in the same format, using the same standard routines constructed from particularly simple and basic language elements. Apart from this, the programs in each group are as far as possible independent of each other and do not use facilities which are tested in other groups. Simple functional tests rely on a small set of basic language elements. All features included in interaction tests are also tested by themselves. Thus each part of the test suite relies only on a small set of basic elements or on those elements and others which can be tested in

isolation. The tests are not, however, arranged in a single sequence. Within a test program the individual tests are generally independent and, except in a few special cases, any test may be deleted without affecting the validity of the remaining tests. The standard output from the tests is arranged to make apparent the identity of any test which has been deleted and to print the totals of tests deleted or failed for each program.

The test suite is distributed as a single file, the population file, on a magnetic tape. This file contains the audit routines themselves, associated test data, and an executive routine, known as VP, which extracts the individual audit routines from the file and tailors them to the environment in which they are to be run.

By means of parameter cards the VP can be directed to extract individual programs or groups of programs from the population file; complete implementer-defined entries in the manner necessary for the system which is to execute them; and generate the job control statements necessary to compile and execute them. A full description of its operation is contained in [NTIS79]. The VP routine is the largest program in the validation system and any computer system which can execute it should have no size-related problems in compiling or executing the audit routines. For some very small systems the size of the executive routine has proved a problem and some effort is going into reducing it. If the VP is too large to be run on the system under test, it may be run on some other system and the tailored audit routines transferred from there to the test system. Other solutions to the problem are possible. The largest of the audit routines proper contains just under 3,000 lines of code, which should be well within the capacity of a production-standard compiler. The size of numeric data items in Cobol is from one to 18 decimal digits. The test suite explores this range fully. The standard makes no explicit provision for floating point data types, but an implementer is free to use floating point arithmetic in the evaluation of arithmetic expressions. For reasons discussed above the test suite does not address this area.

The FCTC tests the functioning of the compiler when presented with programs which conform to one of the four levels of Federal Standard Cobol. When a compiler is first submitted for validation it is put on the list of approved compilers and on the annual validation schedule. Even if the compiler fails some of the tests on its first validation, and some fail many tests, it remains on the list for one year. However, all errors disclosed by the tests must be corrected within that year. Each compiler on the approved list must be revalidated each year and is retained on the list only if no errors disclosed in previous validations remain uncorrected. New errors are permitted, and again the implementer is given a year in which to fix them.

The requirement for revalidation may be waived by the FCTC if the compiler failed no tests at the previous validation; the test suite has not been changed in the interim; and the compiler vendor certifies that neither the compiler nor its supporting operating system has been modified since the previous validation.

The audit routines are fully constrained by the ANSI Cobol standard and the levels of it selected as Federal Standard Cobol. The standard does not prescribe error messages, so no programs in the suite contain syntax or semantic errors. The standard does prescribe the action of a Cobol program

on the detection of certain exception conditions which in some contexts would be regarded as semantic errors. In all cases where it is possible to generate the exception condition predictably by program action, the test suite includes appropriate tests.

Testing is purely functional and there is no attempt to measure any other aspect of the compilers, such as compilation speed or treatment of errors. Compilers which prove to be satisfactory in the functional validation, and thus appear on the Certified Compiler List, can be considered by Federal agencies when they are procuring a compiler. Performance and other non-functional aspects of the compiler will generally be considered by the procuring agency or its advisers at that time. It is Federal policy to use only the standard language, so a point is made of not exploring any possible language extensions.

All the tests are predetermined and remain the same for all validations. It is possible that in future the values of operands in some tests will be changed from release to release, but this is the limit of randomness envisaged.

The Cobol language is well known for the ambiguities in its specification. In general, areas of real ambiguity are not tested and tests are sometimes deleted from the suite when a vendor produces a convincing counter-interpretation of the relevant part of the standard. This does not happen often. For some ambiguities, ANSI X3J4, the committee which is responsible for the standard, produces interpretations stating what was intended. These interpretations do not modify the standard but are generally accepted as a basis for tests in the audit routines.

Most of the programs in the test suite are self-checking. The result of each test, be it a value or a choice of control path, is predetermined. The program can then print a confirmation that the test has been passed, or a message that it has failed and appropriate diagnostic information. At the end of each program the number of failures and the number of tests deleted, if there are any of either, are printed. This makes the checking of most programs much simpler than it might be, since failure messages are very obvious and there is no need to compare thousands of pairs of character strings in case there are one or two mismatches amongst them. This technique is not applicable to all tests and there are some, for example those concerned with page formatting facilities, where the printed output must be examined in detail.

The test suite has evolved over a period approaching 20 years. There are currently four people working full time and about as many again devoting some of their time to maintenance and development of the Cobol and Fortran test suites. Development of version 3.0 of the Cobol tests from version 2.0 took about 1,600 man hours. The test suite is not formally verified, but individual programs are examined by programmers other than their authors for completeness and consistency with the relevant parts of the standard. It is always open to an implementer to object to a test which his compiler fails when he thinks it should pass.

The development of the test suite has been funded completely by the United States Federal Government.

The steps of a formal validation which require access to a computer can be done in a single day on a large fast computer, or about three days on a small system. In both cases, a successful validation with almost all programs compiling first time is assumed. If some programs need modification to make them compile the elapsed time can be much greater. Prior to running the tests, the implementer is expected to determine how the programs need to be tailored to run with his operating system. He must demonstrate that he has done this by running a few specified programs and sending the output to the FCTC. This is accompanied by copies of the relevant compiler and operating system manuals so that the FCTC staff can determine that the programs were run correctly. During the validation itself, the full process of extracting the test programs from the population file, compiling them and executing them is carried out in the presence of the FCTC staff. They inspect the output and ensure that all programs have been run correctly. If any programs need modification, the necessary changes are discussed with the vendor's staff, and all changes to the programs made through the VP routine. Any comments by the vendor's staff on tests which have failed are noted and the tests may be run again with program or job control changes. All the output is taken or sent to the FCTC where it is used as the basis of the Validation Summary Report (VSR). For a perfect validation, the VSR can be produced in less than one working day. For validations with errors it takes longer. The FCTC aims to send a draft VSR to the vendor for comment within two weeks of receipt of the validation output. The vendor is given an opportunity to comment on the VSR before it is published. Any disagreements on its contents may be taken to the Federal Cobol Interpretation Committee for resolution.

All Validation Summary Reports are published and may be obtained from the United States National Technical Information Service, who also sell the official documentation of the test suite [NTIS79].

Formally, the validation is carried out by staff members from the FCTC. In practice the FCTC representatives are observers and the tests are run by the staff of the installation where the validation is performed. This is normally, but not necessarily, operated by the implementer of the compiler being tested. The FCTC does not have its own computers and access to a suitable system must be provided by the body requesting the validation.

The body requesting the validation pays the full cost of the validation, including all computer time used; travel and subsistence for the FCTC validation team; and the time of the validation team on work directly related to that validation. There is an additional standard charge designed to cover computer time used in preparing a copy of the population file for the validation, preparing and checking ASCII input and output files and preparing the VSR for publication. The charge made by NTIS for copies of the VSR covers only the marginal cost of making and mailing an extra copy.

Compilers are tested as an aid to ensuring program portability. The US Federal Government procures computers from a large number of different sources and sees great advantages in being able to run the same program on several different computers. On the whole, the support of the standard by compilers offered for validation has increased with time. On that basis validation of Cobol compilers has been valuable to all parts of the data processing community which seek to move programs from one computer system to another.

The Cobol audit routines have evolved over a long period of time. The style of testing and the organization of the routines has been refined over the years, and any new system would probably be very like the existing suite. The FCTC Fortran audit routines are much more recent, but have been written by the same team in a very similar style.

The same code is used to produce the report from each audit routine, and a library system is used to merge this with the code specific to the particular routine. The style of each test is the same and test code relies on standard routines in the report code to produce pass or fail reports. Each individual test is written to the same structure, with standard support code around it, but no attempt has been made to derive tests automatically from the language description.

Most of the test programs are self-checking. They produce positive confirmation of tests passed but no details of the test. If a test fails, the actual and expected results are printed. In both cases the location of the program listing is indicated by reference to a paragraph-name.

The reporting mechanism works fairly well and the error reports stand out in the test output. However the volume of output continues to grow as new tests are devised and some way of compressing the output still further may have to be found.

As indicated above, the test suite is still being extended. The basic features of the language are fairly well covered, but there is still a lot of scope for testing interaction between them. At present the tests are almost all positive, in the sense that they check that the combination of compiler and execution system does the right thing with well formed and well behaved programs. Another line of development being considered is negative testing, exploring the effect of violating explicit prohibitions of the standard.

A new Cobol standard is due for approval in the first half of 1981. The FCTC is due to begin work in early 1981 on a test suite for this standard. The new standard is expected to contain several new features and development of full audit routines for it will probably extend over a period of years.

Coral 66

(Written by T A D White, Royal Signals and Radar Establishment)

History

In 1964 three parallel but independent activities concerned with the use of high-level languages for real-time applications were taking place within the Ministry of Defence (MoD): the Royal Radar Establishment (RRE), now the Royal Signals and Radar Establishment (RSRE), was acting as the research and design authority for a large real-time computer project, which had required the design of a new language; secondly, an MoD working party was examining the future programming requirements of the Ministry and had been asked to formulate a policy for the use of computers in defence applications - it was at this time that the suggestion of a standard high-level language for use throughout the Ministry was made; thirdly, at RRE, Malvern, a separate research group was publishing the results of its work into techniques for the automatic generation of compilers. Thus, by the end of 1966 there were three essential ingredients necessary for the development of a high-level language for use in military systems and for its standardization across the Ministry of Defence user community. The need for a language control authority which had the power to approve or recommend an implementation of a Coral 66 compiler for Ministry use was obvious, as, indeed, was the need for an evaluation mechanism on which the approval could be based. The Inter-Establishment Committee for Computer Applications (IECCA) took responsibility for the Coral 66 language and tasked the Computer Applications Division of RSRE to provide an evaluation procedure.

Machine dependent features of Coral 66

Coral 66 was designed in response to the need for a high-level language on small (1960s) mini-computers in a control environment performing tasks which had previously been implemented almost without exception in assembly code or machine code. In particular Coral 66 was required to produce efficient code and be allowed to exploit directly special or particular hardware features and thus it was felt that some debasement of high-level language ideals was acceptable.

Coral 66 may be regarded as a kernel language, which defines those features which will always be present in a Coral 66 implementation, and a set of optional major features. A Coral 66 compiler which implements all optional major features is referred to as a "full Coral 66" compiler. In addition, Coral 66 defines facilities to enable the underlying hardware of a particular Coral 66 machine to be exploited. Further, certain operations are defined in terms of the machine's behaviour: for example, the rounding algorithm is to be that of the hardware.

Evaluation of a Coral 66 machine

IECCA had undertaken neither to certify potential Coral 66 implementations nor to validate them : it had undertaken to provide advice on whether a Coral 66 compiler was suitable for MoD use or not, and if so to acknowledge it as being on the "IECCA list of standard Coral 66 machines approved for MoD use". Acting for IECCA, RSRE designed, and is still evolving, a suite of compiler test programs which exercise various Coral 66 language features to demonstrate their conformity to the *Official Definition of Coral 66*. It may be argued that this method of demonstrating conformity is closely akin to validation : a Coral 66 machine is, however, at best "approved for MoD use".

After Coral 66 had become the "preferred" high-level language in accordance with MoD computer policy, IECCA envisaged that hardware manufacturers would provide Coral on their machines and seek approval in order to become eligible for MoD contracts. Thus, in the interests of standardization across the Ministry, MoD would fund the evaluation of Coral machines offered by hardware manufacturers for approval.

IECCA have always regarded the evaluation as being addressed to the Coral 66 machine, that is the hardware on which a Coral 66 compiler runs, the compiler itself, the hardware to which a Coral 66 object program is targetted, and the host/target program development/support environment. However, it is only of late that the importance of the environment has been fully appreciated. In the absence of formal requirements for a Coral 66 program support environment the evaluation of a Coral 66 machine includes a subjective report by the evaluator on the tools provided by the support system. The remainder of these notes confine themselves to evaluation of the Coral 66 compiler itself.

RSRE Coral 66 compiler test suite

The RSRE Coral 66 compiler test suite was originally designed by members of RSRE's Computer Applications Division and has been evolving since 1974. It generally assumes that it is possible to reserve Coral INTEGER storage space (declarations), manipulate simple INTEGER quantities (correct interpretation of integer denotations and assignment to integer variables), and call procedures with few parameters each of which is a simply written expression. The tests also assume that certain machine dependent input/output is provided : there is a compiler test to demonstrate that the input/output required by the suite is provided satisfactorily.

The tests use these assumptions to exercise the remaining language features of Coral 66 - the optional major features mentioned earlier (such as recursion, Coral 66 tables, Coral 66 data overlaying, Coral 66 bit manipulation) and the *kernel* features expected in any Coral 66 implementation appearing in a variety of involved pieces of text (eg deeply nested conditional expressions, loops with involved *for-lists*). Machine dependent features are tested either by the evaluator being required to write a machine dependent portion of Coral text with a given functionality for inclusion in a test, or by the *preferred* interpretation(s) being coded into the test and the *actual* test results being compared against results expected from the *preferred* interpretation(s). There is one test which allows comment on a compiler's ability to detect and recover from errors. A further test investigates the arbitrary limits imposed by a compiler on a Coral 66 program.

The format of the results produced by the tests differs from test to test, but is always described in the operating instructions which are included in each test as a Coral 66 comment. Generally, however, the results require the generation of a message or sequence of codes. Error codes corresponding to individual Coral statements indicate a deviation from the expected behaviour.

The tests establish the degree of conformity of the candidate compiler : and as such is not a simple pass/fail - IECCA allows certain freedom in the interpretation of some Coral 66 features. A set of benchmark programs allows comment to be made on the efficiency or cost of a Coral construct.

The tests themselves have been analysed by a program derived from the research into automatic generation of compilers – the SEMSID driven test-tester confirmed that every construct appeared in every syntactically legal position.

Formally, Coral 66 does not admit to extensions. However, it is known that there are many implementations of Coral 66 which consider data objects of size different from the single Coral INTEGER word – the so-called *byte* objects and *long* objects. Similarly there are many implementations of shifts. A manufacturer offering a candidate compiler would be expected to reveal *enhancements* and the evaluator would be expected to satisfy himself that the implementation of the *enhancements* does not clash with any other feature and is *within the spirit* of the Official Definition.

Management of an evaluation

It is expected that a hardware manufacturer wishes to provide Coral 66, the MoD preferred high-level language, on his machine to enable him to be eligible for MoD contracts. Once RSRE has been approached by the manufacturer and invited to test the candidate compiler, a contract is let to a third party (usually a software house). The candidate compiler is then tested against the compiler tests and measured against the benchmarks : a period of about 2 weeks is assumed for this. The evaluator is also expected to satisfy himself that *extensions* are within the *spirit of Coral*. The evaluator spends approximately four weeks interpreting results and preparing a report detailing the facts of the compiler's performance and his subjective opinions of the program support environment. The report is then received for discussion by IECCA – it is at this time that the manufacturer may be requested by IECCA to rectify shortcomings in his compiler or support environment. A report, once accepted, is made publicly available and the compiler placed on the IECCA standard list.

An evaluation costs about £3000, but this does not include any contribution
to producing the validation suite or the indirect costs incurred by the manufacturer of the compiler.

Retesting of Coral 66 compilers

No formal retesting of Coral 66 machines is carried out by the Ministry. However, the Ministry feels that it is at liberty to require an informal run of the RSRE test programs whenever a new version of a compiler is issued or when the underlying machine is altered. The manufacturer would be given chance to make good any deterioration in performance, if any, noted during the informal run of the test program. Failure to repair degradations could result in the Coral 66 machine being removed from the Standards List.

Fortran

Both the US Navy and the National Bureau of Standards have produced suites of programs for validating Fortran compilers. The two suites are described in reports by Holberton and Parker [HOLB74] and Hoyt [HOYT77]. In both suites the aim is to see whether a compiler conforms to the American Standard Fortran (X3.9-1966), i.e. whether it correctly processes programs that conform to the standard. As far as possible each feature is tested in

isolation. The tests are comprehensive in testing each feature of the language but do so only cursorily, for example, the function 'EXP' is tested in the NBS suite by checking that the result is correct for seven different arguments.

The limited objectives mean that the tests do not necessarily distinguish between an excellent compiler and a very bad compiler; a compiler may pass all the tests and yet be absolutely useless for all practical purposes, for example it might be too slow, or it might need too much memory, or it might give no help in detecting program errors.

The tests also make no attempt to discover the attitude taken towards extensions to Standard Fortran, for example, what extensions have been provided, can a programmer be informed when he uses a non-standard facility, is there a way of converting a program with extensions into Standard Fortran?

The NBS and US Navy suites differ in the way they report the results of testing a compiler. Hoyt [HOYT77] describes the differences:

> "Output results [in FCVS] produced by the execution of each routine indicate whether the code generated by the compiler passed or failed each test of the routine."

On the other hand, a major flaw in the NBS suite is that

> "all the test results were listed on a printer and required careful examination ... by the user."

The NBS Fortran suite

This test suite was developed over several years and released in 1974. A basic assumption is made that the compiler is working but that not every feature has been implemented. The test suite contains more than 14000 cards.

The tests are now obsolescent; a new Fortran standard has been published (Fortran 77, [ANSI78]) and many of the tests will have to be revised and further tests added.

FCVS - The US Navy test suite

This suite was started after the NBS suite. The project was first designed in 1973 but remained in abeyance until February 1975 when the decision was made to build a self-measuring suite which would adequately test all the elements of the Fortran language specified in the 1966 standard. The project started in October 1975, but in March 1976 the objectives were changed to test

> "the conformance of those elements of the Fortran language which are contained in the logical intersection of the American Standard Fortran, X3.9-1966, and the elements proposed for the subset language in the draft proposed American National Standard programming language Fortran."

The FCVS (i.e Fortran Compiler Validation System)

> "consists of Fortran audit routines, their related test data, and an executive routine (EXECUTIVE) which prepares the audit routines for compilation and execution."

The EXECUTIVE reads command lines, typically from a card reader or

online terminal in order to compile and run any desired set of audit routines. Facilities include the ability to insert job control instructions before, inside and after each test program, to alter the file of audit routines to correct errors in them, and to remove audit routines that failed to compile in earlier tests.

The audit routines are written in a very restricted form of Fortran so that they will run on any Fortran system.

"Only the simplest forms of GO TO, Arithmetic IF, WRITE, and assignment statements are used to write the support code required for each test."

The statement tests are also built from a similar

"basic set of Fortran language features which are assumed to function correctly. The remaining language features are tested using these basic language elements." ... "The first several routines in the FCVS test the language elements in the basic assumptions. Their correct execution ensures that the failure of any test in the remainder of the routines is due to the improper implementation of the language feature being tested."

Hoyt states

"The tests in the FCVS are 'positive' in that only statements permitted by the Standard are included. There are no 'negative' tests of incorrect statement formats which a compiler is supposed to flag as errors.

The FCVS also does not test vendor extensions to the language specifications, and does not perform an error analysis on the results of executing the Basic External Functions supplied by Fortran processors. The FCVS is not designated to measure the efficiency of the object code generated or the performance characteristics of a Fortran compiler."

The FCVS has been extended for Fortran 77. The first version in 1978 tested the subset level of standard Fortran 77, its use is described in [FCVS78]. However,

"major additions to the current FCVS will be required to test the new language features in the revised Standard. The motivation and philosophies previously described for the current FCVS remain essentially intact in developing a compiler validation system for the complete revised language Standard."

"The FCVS will be used by the ADPE Selection Office, Department of the Navy, in the procurement process. It is an important addition to procurement procedures and the FCVS will ensure the selection of computer systems with compilers that support the Fortran standard".

The Fortran tests are now administered by the Federal Compiler Testing Center, and they have been validating against the 1978 Fortran standard for some time. Their 1980 validation schedule contained six validations against the 1978 standard and only two against the 1966 standard. However in October 1980, the tests check only features in the subset level of Fortran 77.

A report is published on each Fortran compiler tested which can be obtained in Britain on microfiche from the British Lending Library. Some of the results and more details are given in Section 7.4.

Jovial

Jovial is an American language which is used mainly by the American Department of Defence. When DoD require a computer with a Jovial compiler they ensure that the compiler is satisfactory by insisting that it pass certain tests. It is thus comparable with Coral in the United Kingdom. There are other similarities, Jovial also dates from the 1960s and is based loosely on Algol [The name was originally an acronym = Jules' (Schwartz's) Own Version of the International Algebraic Language, i.e. Algol 58]. Some parts of Jovial are not defined in the standard so that they can be implemented efficiently on each different computer system. These computer dependent features hinder standardization.

The information below is taken from a report [ROBI73] which gives full details of the test suite, its use and history.

The test suite was originally developed in 1968; this first version aimed at checking that the various language features all compile and execute correctly. The basic notions of the language are assumed to be correct and are not tested explicitly. Each test program uses the basic notions to check one additional facet of the Jovial language. Each test module is written to be machine independent, and the test results are designed to be easily readable so that it is easy to check that the compiler has worked correctly. There is a logical difficulty writing the tests because the Jovial standard only defines very primitive input/output. All implementations have different I/O facilities which naturally makes it difficult to output results from the test suite; in practice it is necessary to write an extra module for each system before running the test suite. The report specifically mentions the difficulty of producing tests containing all combinations of data defining attributes.

Extra tests were later added as deficiencies became evident; these were of five sorts:

(1) To check that syntax errors are detected by the compiler. The evaluation process for these tests are tedious because it is necessary to recognize and note the absence of a message. It is also necessary to examine the compiled object code to confirm that the error correction is reasonable.

(2) To check that the compiler can compile and run programs of the size required by the procuring agency. These extra tests are written by the procuring agency.

(3) To check the compiler's 'efficiency', i.e. measure the average number of machine instructions compiled for each Jovial statement.

(4) To test the various machine dependent features allowed in Jovial programs.

(5) To test known trouble spots and features that had caused errors in earlier compilers.

LTR

LTR is a real time language used mainly by the French Ministry of Defense, but also by about 50 other industrial installations. CELAR (Centre Electronique de l'Armement) have a certification facility and all contractors with MOD are required to conform with a Standard Compiler.

Pascal

Editorial note: The other chapters of this book give a fuller account of the Pascal compiler validation suite and so this section has been deleted.

Pearl

Pearl (Process and Experiment Automation Realtime Language) is a language designed and implemented in Germany. P Hruschka summarizes the validation and standardization that has taken place in [NCC 80].

A group at the University of Stuttgart have developed a validation suite for Basic Pearl (a standard subset).

Pearl is defined using Attribute grammars which can express context dependent features of programs as well as the simple syntax. The semantics are expressed using Petri nets and technical English. It is an important advantage of the language definition that it provides the basis for a compiler.

RTL/2

(Written by J G P Barnes, SPL International)

All commercially available RTL/2 compilers use the same front-end and so the testing of new compilers has naturally been concentrated on the testing of back-ends or code generators. (Other RTL/2 front-ends have been written but for various reasons have not been made commercially available).

The RTL/2 front-end was written in 1972 and although minor changes have been made it has not been seen necessary to create a formal testing mechanism aimed at the front-end.

In order to simplify the testing of back-ends a standard test suite was written in 1976. It has been added to in the light of experience.

This test suite consists of some 50 program fragments which can be combined together to form one or more complete executable programs. The suite has also been designed so that it can include its own internal output routines or can be linked to the standrad system routines. This has been done so that the program can be compiled as a single monolithic program and reliance on the linkage system is therefore minimized.

RTL/2 is a closely defined language [BSI 80] and there is little freedom for variation in interpretation. Accordingly there is no need for the tests to allow for language variation.

All tests behave in the same way – they compute one or more results by two routes and then compare the values. In the case of real arithmetic the allowed tolerance is given by the user, together with other machine parameters in a data brick in one of the program fragments.

Each group of tests cycles over a set of data which includes likely pathological values. If all the tests of a group pass then only a confirmatory message is output. If any test fails then a more detailed description is provided.

The user has additional control over whether overflow detection is to be used or not. He can also easily single shot a test at the console of a mini-computer; there is a facility to halt the computer just before a specific test.

The tests have been arranged sequentially so that later ones assume that some earlier ones are successful. If basic things such as integer arithmetic do not work then the error reporting routines will not work either. A special initial group of tests is available which should fail and therefore exercise the error reporting routines and data selection routines over the full set of data.

The tests cover all aspects of the code generation of RTL/2 and run time system including such things as array bound checks, stack overflow and the standard error recovery mechanism. But of course some combinations of optimizations on a strange machine may not be specifically tested. Interactions between unusual optimizations should be checked for separately.

The tests do not however cover all aspects of RTL/2 itself – it really only tests the instructions as seen by the intermediate language. Thus there are no checks on WHILE since this is merely passed to the back-end as simple tests and jumps. Similarly there are few explicit tests on fractions since they are treated identically to integers in most cases.

The tests are quite searching. When first written they were applied retrospectively to compilers that had been in wide use for some time and odd errors in these were revealed.

6. Summary of existing validation suites

Approximate size:
Basic (Version 1): 161 programs
Basic (Version 2): 208 programs
Cobol (in 1969): 18 000 lines
Cobol (1978, Version 3.0 for the 1974 standard): 218 000 lines
Fortran (NBS): 14 000 lines
Pascal (version 3.0): 20 000 lines
Pearl Subset: 20 000 lines (of which 6000 test realtime facilities)
RTL/2: 50 programs

Test strategy
Incremental (i.e. each feature is tested before being used in other tests): Basic, Pearl Subset, RTL/2.

Basic features are assumed to be working:

Coral, Fortran (NBS), Fortran (FCTS).

Each feature tested in turn:
Ada, Pascal.

Invalid programs are tested:
Ada, Basic (Version 2), Jovial (syntax errors only), Pascal.

Approximate effort to produce validation suite:
Basic: 3 man-years for version 1, a further 15 man-months for version 2.
Cobol: about 20 man years up to 1980; about 1600 man hours to develop version 3.0 from 2.0, four full time staff in 1980.
Pascal: about 7 man years (up to June 1982).

Cost of validating a compiler:
Coral: £3 000 paid by MoD to contractor.
Fortran and Cobol (FCTS): about $7 000 is charged to the manufacturer, this covers the direct validation costs and represents about half the total operating cost.

A validation report is published:
Fortran (FCTS), Cobol (CCVS9), Coral.

Date when the validation suite was written:
Basic (Version 1): 1975-77.
Basic (Version 2): 1979.
Cobol (FCTS): first released in 1968.
Coral: 1974 onwards.
Fortran (NBS): released in 1974.
Fortran (FCTS): 1975 onwards.
Jovial: 1968 onwards.
Pearl Subset: 1976 onwards.
Pascal: 1977 onwards.

Revalidation.
Coral: informally when thought necessary .
Fortran and Cobol (FCTS): annually.

7.1 Testing floating-point arithmetic

One non-trivial validation problem is testing a compiler's numerical properties. The best that can be hoped for from any compiler is approximately correct results and a compiler validation suite cannot just report pass or fail but must specify the accuracy of the operations.

Floating-point operations and standard functions should thus be tested in depth. Part of the tests should perform fixed specified operations so that the results are directly comparable. However additional pseudo-random tests would prevent compilers from being tuned to give abnormally good results for the test suite.

The facilities that must be tested in this way include:

Input/output.
Input and output a decimal value, in both fixed and floating-point formats.

Floating-point operations.
Arithmetic operations, add, subtract, negate, absolute value, multiply, divide, exponentiate with integer exponent, exponentiate with real positive exponent.

Standard functions.
sine, cosine, exponential, logarithm, arctangent, etc.

The results must be reported in several ways:
The bias, i.e. whether computed values are normally too big or too small.
The usual error, i.e. the value such that half of all computed values have an error less than this amount.
The normal maximum error, i.e. the value such that 95 per cent of all computed values have an error less than this amount.
The most extreme error recorded.

For arithmetic operations, the most meaningful definition of error will normally be the correct number of significant digits, with some variation when the computed value or result is near zero.

The aim of these tests is mainly to measure the performance of a compiler, but some results should be regarded as failing the tests, for example:

Outputting a value that is incorrect by more than one in the last decimal place.

Inputting a value that is more than one bit wrong.

Producing a negative result for any of the following functions:

```
real procedure abs(x); value x; real x;
real procedure sqrt(x); value x; real x;
real procedure exp(x); value x; real x;
```

Producing a value outside the closed interval [-1.0, 1.0] for any of the following functions:

```
real procedure sin(x); value x; real x;
real procedure cos(x); value x; real x;
```

Implementing these tests

The tests described above can only be performed if it is known exactly what values are being used inside the computer. Thus it is necessary to be able to input and output the actual binary digits that represent a floating-point value inside the computer. In practice it is more convenient to read and print octal or hexadecimal digits. If the test program also reads the exact result in the same way, it is possible to compare it with the computed result, calculate the difference, and report on the compiler's performance.

Inside the test program, each octal value would be stored exactly as a record with the structure:

```
RECORD positive, exponentsign : boolean;
```

```
exponent : ARRAY [1 ..3] OF octaldigit;
mantissa : ARRAY [1 .. nn] OF octaldigit END
```

Procedures for the following operations would be needed:
read octal value,
print octal value,
standardize octal value,
convert octal value to real,
convert real to octal value,
add two octal values,
negate octal value,
multiply two octal values,
divide one octal value by another,
abs of octal value

Note that this method, unlike that suggested in [BAIL77], is applicable to double-length arithmetic and functions, and can also be adapted for complex arithmetic.

7.2 An anoimous Babel program

This program is written in Babel and compiled on a KDF9 at NPL. It demonstrates an odd feature of integers: one particular value could not be assigned to a variable. The effect never caused trouble in user programs; and it is unlikely that it would ever be discovered treating the compiler as a black box, yet in some senses this feature of the compiler is an error.

One particular value cannot be assigned to a variable

One particular integer value cannot be used in calculation because the compiler thinks the value is 'undefined'.

```
:BLOCK
    :INTEGER ii, jj, kk
:DO
    jj := 140 32800 * 100 00000 + 69 42718;
    out li(jj);
    kk := jj + 2;
    out li(kk);
    ii := kk - 1;
    out li(ii);
:ENDBLOCK :EM
```

The result of compiling and running the program

```
BABEL RUNTIME FAILURE BETWEEN LINE 9 AND END
VARIABLE USED IS UNASSIGNED

        THE RETROACTIVE TRACE

   OUT LI     2
```

```
THE START OF THE PROGRAM


        THE VALUE OF ALL THE ENTITIES

NOTE

AN   A   PRINTED AFTER THE VALUE OF A VARIABLE INDICATES THAT
IT HAS NOT BEEN USED SINCE THE LAST ASSIGNMENT TO IT

THE PROGRAM FAILED IN BLOCK STARTING NEAR LINE 2
 IDENT    TYPE   MODE          VALUE
       II INT          UNASSIGNED  A
       JJ INT           14032 80069 42720
       KK INT           14032 80069 42718

CALLED FROM GLOBAL BLOCK.

          A FLOWTRACE OF THE PROGRAM

   LINE   PASSED     DESCRIPTION

      1         1              BLOCK


          END OF FLOWTRACE
```

7.3 Validating process control systems

Z J Ciechanowicz and R S Scowen attended a one day British Computer Society conference (6 Feb 1980) on "Security, Reliability and Integrity in Process Control Systems" to see how many of the techniques used to validate process control systems could be applied to the problems of compiler validation. There were four speakers, two on the control of atomic reactors, one on railway signalling, and one on air traffic control.

A key factor in achieving safety and reliability is **redundancy**, i.e. extra equipment so that no one item is essential to the correct working of the whole system. However duplication makes it necessary to consider and prevent error propagation: a broken module must not corrupt or otherwise upset a module that is functioning correctly.

There are two different reactions to errors, a system must be **failure tolerant**, i.e. it must continue to be available even in the presence of broken or malfunctioning modules. A system must also *fail safe*, i.e. it must shut down safely as a last resort.

Two different techniques that ensure reliability and safety are continuous monitoring that all is still well, and using simple and unadventurous methods to implement the system.

It is not generally useful to have independently written software modules. It is too wasteful of programming resources and does not prevent problems caused by specification errors. Instead programming problems are minimized

by various techniques including:

Refine the specification iteratively.

Design all software formally.

Use program analysis to prove properties of the program.

Test the program with carefully chosen data.

Although there is a well-established methodology for predicting the frequency and sort of hardware errors that will occur, there are hardly any figures that can be used to predict software performance.

Professor Randell (Newcastle University) commented that trying to validate any software by treating it as a black box, i.e. looking only at the output from various test cases, should be thoroughly discredited by now. Yet this is precisely the technique used in all current compiler validation packages. He also compared a constant validation suite with the practice of setting the same examination questions every year. While ruefully admitting the truth of these statements we must repeat that this is the only economically viable method that is currently available. Further, validation suites can be based on past experience, e.g. test for the existence of faults that have previously occurred, and use the knowledge of how compilers are constructed. Validation suites can also be continually extended so that although there can be no certainty of detecting all errors, we can at least ensure that they do not recur.

Another speaker mentioned that hardware reliability typically improves with age, most errors are found in infancy. The same is undoubtedly true of software. Another similarity is that modifications (euphemistically called maintenance or enhancements) often introduce errors. This strongly suggests that a compiler should be revalidated after any amendments and that there should be some method of ensuring that the compiler in use is identical with the one that was validated.

Professor Pyle (University of York) mentioned the technique of preventing catastrophic errors by monitoring the frequency of *near misses*. The idea is that a single isolated error can probably be foreseen and safely dealt with; catastrophes are unlikely except when two independent faults occur simultaneously and interact in an unforeseen way. The technique enables the probability of catastrophes to be estimated more accurately than using the data of actual disasters – rare and random events.

7.4 The results of validating compilers

This appendix summarizes very briefly the published results of various compilers. Of course, different validation systems may give different results for the same compiler. The full references for the various validation reports are:

FCVS
Fortran Compiler Validation Summary Report FCVS66-VSRnnn,
Federal Cobol compiler testing service, Department of the Navy, Washinton DC, USA 20376.

Honeywell System Level 66, Fortran compiler Release 3I, GCOS 3I operating system,
using tests FCVS66 version 1.1.
Ref No FCVS66-VSR190 (AD A040 083).

No errors were detected but the report specifies the action following PAUSE integer and STOP integer statements.

The compiler printed warning messages telling the programmer:
(1) Some labels appear in an ASSIGNED-GO-TO statement but not in an ASSIGN statement.
(2) Comparing two REAL values for equality may not be meaningful.
(3) Format specifications such as F3.2 have incompatible W.D field.

UNIVAC 1100/43, UNIVAC ASCII Fortran Release 6R1 compiler, EXEC 33R1 operating system,
using tests FCVS66 version 1.2.
Ref No FCVS66-VSR200 (AD A040 355).

The compiler failed to translate a statement with 57 nested parentheses.

```
IVCOMP = ((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((
          IVON01 / IVON02
          ))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
```

No other errors were detected but the compiler printed warning messages telling the programmer:
(1) "STMT FUNCTION NAME '<var name>' IS PREVIOUSLY DEFINED".
(2) "VARIABLE '<var name>' MAY BE USED BEFORE SET TO A VALUE". The latter message occurred only when the FCVS was run with the optimization feature requested.

IBM System 360/65, Fortran CODE and GO Release 3.0 compiler, OS 21.8 MVT with HASP 3.1 operating system,
using tests FCVS66 version 1.2.
Ref No FCVS66-VSR215 (AD A040 128).

No errors were detected but the compiler printed warning messages telling the programmer:
(1) that a blank source line is an "ILLEGAL STMT".

Burroughs B7700, Fortran II.9 compiler, MCP II.9 operating system,
using tests FCVS66 version 1.2.
Ref No FCVS66-VSR255 (AD A046 081).

The compiler failed to translate a statement with 57 nested parentheses.

```
IVCOMP = ((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((
          IVON01 / IVON02
          ))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
```

The report adds a note that although this statement failed with the

default optimization setting (OPT=0), it compiled successfully with (OPT=1).

The report also specifies:

The action following PAUSE integer and STOP integer statements.

The compiler printed warning messages telling the programmer about a label that appears in an ASSIGNED-GO-TO statement but not in an ASSIGN statement.

IBM System 370/168, Fortran IV H Extended Level 2.2 compiler, MVS, Release 3.7, JES2 operating system,
using tests FCVS66 version 1.2.
Ref No FCVS66-VSR260 (AD A048 353).

The compiler failed with two statements printing a message "ALL THE ARGUMENTS OF AN ARITHMETIC STATEMENT FUNCTION ARE NOT USED IN THE DEFINITION".

IBM System 370/168, Fortran IV G1 Release 2.0 compiler, MVS, Release 3.7, JES2 operating system,
using tests FCVS66 version 1.2.
Ref No FCVS66-VSR265 (AD A048 317).

The compiler failed to translate a statement with 57 nested parentheses.

```
IVCOMP = ((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((
          IVON01 / IVON02
          ))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
```

A completely blank line also caused the compiler to print an error message .

Control Data CDC Cyber 174, Fortran IV Release 4.6 compiler, NOSBE, Release 1.2 operating system,
using tests FCVS66 version 1.2.
Ref No FCVS66-VSR300 (AD A053 035).

No errors were detected and no warning messages were produced.

Control Data CDC Cyber 174, Fortran IV Release 4.6 compiler, NOS, Release 1.2 operating system,
using tests FCVS66 version 1.2.
Ref No FCVS66-VSR305 (AD A053 055).

No errors were detected and no warning messages were produced.

Control Data CDC STAR 100, Fortran IV Release 1.3 Cycle 13 compiler, STAR OS, Release 1.3 operating system,
using tests FCVS66 version 1.2.
Ref No FCVS66-VSR350 (AD A064 668).

No errors were detected and no warning messages were produced.

7.5 The value of warning messages

When computer scientists talk of debugging programs they mean detecting and removing the errors. This naturally encourages an over-simple view that a program is either right or wrong. This appendix suggests that there are at least seven degrees of correctness for a program and uses as an example a simple Fortran 77 program that calculates the average of up to ten numbers.

(1) NONSENSE. These programs are not syntactically correct and any compiler is sure to fail during translation, e.g.

```
      PROGRAM  AVERAGE

      REAL       AA[10],  SUM
      INTEGER    II, NN

      READ (5, *) NN
      READ (5, *) (AA(II), II=1,NN
      SUM = 0.0
      DO 10, II := 1, NN
          SUM = SUM + AA(II)
   10 CONTINUE
      WRITE (6  *) SUM / NN

      END
```

This 'program' contains various syntax errors:

(a) The bounds of array "AA" should be "(10)".

(b) A comma is missing from the WRITE statement.

(c) The programmer has written ":=" (becomes) instead of "=" (equals) in a DO statement.

(d) A ")" (right bracket) is missing from the second READ statement

(2) ILLEGAL. The program is syntactically correct but performs some undefined action at runtime, e.g.

```
      PROGRAM  AVERAG

      REAL       AA(10),  SUM
      INTEGER    II, NN

      READ (5, *) NN
      READ (5, *) (AA(1), II=1,NN)
      SUM = 0.0
      DO 10, II = 1, NN
          SUM = SUM + AA(II)
   10 CONTINUE
      WRITE (6, *) SUM / NN

      END
```

If "NN" is greater than one, the program ought to fail at runtime because the values of array elements "AA(2 : NN)" are undefined. If the compiler does not fail the program, anything might happen.

(3) ALMOST WORKING. The program sometimes, but not always, gives the correct answer, e.g.

```
      PROGRAM  AVERAG

      REAL       AA(10),  SUM
      INTEGER    II, NN

      READ (5, *) NN
      READ (5, *) (AA(II), II=1,NN)
      SUM = 0.0
      DO 10, II = 1, NN
          SUM = SUM + AA(1)
   10 CONTINUE
      WRITE (6, *) SUM / NN

      END
```

The program prints the correct answer - but only when "NN" equals one, or by coincidence (e.g. when all the array elements "AA(II)" have the same value). Such programs are often described as "90% finished". If they are too big, complicated or badly designed, they inevitably stay that way.

(4) FRAGILE. These programs are syntactically correct and print the correct answer with valid data. But with incorrect data there will be non-standard operations during execution, e.g.

```
      PROGRAM  AVERAG

      REAL       AA(10),  SUM
      INTEGER    II, NN

      READ (5, *) NN
      READ (5, *) (AA(II), II=1,NN)
      SUM = 0.0
      DO 10, II = 1, NN
          SUM = SUM + AA(II)
   10 CONTINUE
      WRITE (6, *) SUM / NN

      END
```

If a zero value is read for "NN" the program will try to calculate zero divided by zero during the WRITE statement. Problems are also likely when there are fewer than "NN" values in the data file, or if "NN" is greater than 10.

This program, being non-standard, is treated in different ways by different compilers. Failures may be reported, on the other hand the program may print 'correct` answers.

(5) SUSPICIOUS. Programs may be syntactically and semantically correct but contain operations which have no sensible effect. Consider another version of the program to calculate an average, e.g.

```
      PROGRAM  AVERAG

      REAL     AA(10),  RT MN SQ,  STD DEV,  SUM,  XX
      INTEGER  II, NN

      READ (5, *) NN
      IF ((NN .LT. 1) .OR. (NN .GT. 10)) THEN
        WRITE (6, *) 'WARNING-INCORRECT DATA FILE, N OUT OF RANGE'
      ELSE
        READ (5, *, END = 99) (AA(II), II=1,NN)
        SUM = 0.0
        XX = 0.0
        DO 10, II = 1, NN
          SUM = SUM + AA(II)
          XX = XX + AA(II) * AA(II)
10      CONTINUE
        RT MN SQ = SQRT(XX / NN)
        WRITE (6, *) SUM / NN
      END IF
      STOP

99    WRITE (6, *) 'WARNING-INCOMPLETE DATA FILE'
      END
```

This program computes the value of "root mean square" for no apparent reason, and never refers to the variable "standard deviation" at all after its declaration. At best the program is unnecessarily big and slow, more likely the programmer has forgotten something.

(6) WRONG. A program may be correct, robust and never fail, but nevertheless be wrong because the customer actually wanted a different function to be calculated, e.g.

```
      PROGRAM  AVERAG

      REAL     AA(10),  SUM
      INTEGER  II, NN

      READ (5, *) NN
      IF ((NN .LT. 1) .OR. (NN .GT. 10)) THEN
        WRITE (6, *) 'WARNING-INCORRECT DATA FILE, N OUT OF RANGE'
      ELSE
        READ (5, *, END = 99) (AA(II), II=1,NN)
        SUM = 0.0
        DO 10, II = 1, NN
          SUM = SUM + AA(II)
10      CONTINUE
        WRITE (6, *) SUM / NN
      END IF
      STOP
```

```
99 WRITE (6, *) 'WARNING-INCOMPLETE DATA FILE'
   END
```

This sort of program is the most dangerous of all. It calculates the average (arithmetic mean) of the values perfectly; nevertheless it is wrong because the customer actually wanted the median of the values. No compiler can be expected to report errors of this sort.

(7) CORRECT. A seventh class of programs, correct in every way is believed to exist by a few computer scientists. However no example could be found to include here.

Conclusions

"Nonsense" programs are most unlikely to cause problems, almost every compiler will report an error during translation. Many compilers will also fault "illegal" programs by making runtime checks that, for example, subscripts are within bounds, and that overflow has not occurred. "Almost working" and "fragile" programs can sometimes be detected and cured by a programmer running carefully prepared test cases (assuming the program is designed sensibly).

"Suspicious" programs are more dangerous; a few compilers warn the programmer of variables that are never used or assigned a value. Less obvious peculiarities are almost never found, thus DAVE, built by Osterweil and Fosdick [OSTE76], is one of the few compiler systems able to detect unnecessary assignments. DAVE analyses a complete Fortran program and warns the programmer of these and other anomalous features. Because DAVE analyses the whole program statically it is unable to distinguish one array element from another and cannot cope with recursive programs. Babel (Scowen [SCOW79]) is another system able to detect suspicious programs. It makes runtime checks and prints an error when a variable is assigned a value that is never subsequently evaluated. DAVE and Babel simplify greatly the production of correct software. For example, at the end of the "almost-working" program given above, Babel reports that the array elements AA(2 : NN) have not been evaluated. And in the "illegal" program, Babel finds the error earlier because it will fail when another value is assigned to AA[1] without the first value having been read.

Of course a "wrong" program cannot be detected at all, it is the likely result of an informal specification. But in all other cases programming is easier and the results are superior if the compiler makes all sensible compile-time and runtime checks, and then reports any deviations clearly. It is a serious fault in Fortran that the standards make no such requirement. A survey of different compilers (see the results in Section 7.4) has shown that although some warn the programmer of certain odd features, none do it consistently.

A compiler is also a program, and the aim of compiler validation is to check that the compiler is correct. The examples above show that this aim cannot be satisfied at present. The best that can be done by running various test cases is to show a compiler is "almost-working". Difficulties are compounded because the specification, i.e the language standard, is usually

ill-defined.

All software would be more reliable if compilers were required to detect and report program errors. Unfortunately the latest programming language standards still take a permissive view. For example the draft standard for Pascal [ISO 80] defines various conditions that should cause a runtime error, but then states that a compiler shall conform to the standard if there is a statement in the accompanying documentation that particular errors are not reported. Thus a compiler could conform to the draft ISO standard and yet only detect the sort of errors in the first program. Ada is a little better. However although the Preliminary Ada Reference Manual [ICHB79] stated that any attempt to evaluate an unassigned variable causes a NO_VALUE_ERROR exception, in the revision [ICHB80] a year later, this fault detection feature has been removed.

One reader of a draft of this report believes the last paragraph is "too one-sided". He commented:

> "There are certain invalid uses of a language (NO_VALUE_ERROR is one) that are too expensive to check for at run-time for many applications, and that increase compiler development time. Similarly, some misuses of language constructs cannot be detected for sure at either run time or compile time (e.g. in Ada, reliance on the order of evaluation of parameters in subprogram calls). The disadvantages in terms of language complexity or run-time inefficiency have been judged to outweigh the advantages of mandating compile-time or run-time error detection. Since compilers cannot in any event detect all programming errors, it is a matter of judgement which errors should be detected by a language and which left to normal software development procedures. The goal of having a language in which there are no "erroneous" constructs (to use Ada terminology) is too costly. A language having no "erroneous" constructs is likely to have many non-conforming implementations, since the run-time cost or the delay in delivering a conforming compiler will encourage deviations from the Standard in these respects. If NO_VALUE_ERROR had been left in Ada, how many military applications would sanction its use? Your example in Section 7.2 also illustrates the difficulties in trying to implement NO_VALUE_ERROR in a space-efficient manner.
>
> In short, the permissive view that should be discouraged is the view that says any program for which the Standard does not provide a meaning is permitted to have an implementation-defined meaning. With such a rule it is impossible to enforce standards effectively."

We recognize the arguments in this protest but believe they are mistaken. The benefits of making the checks are not easily quantified, but they include:

Programs are better designed. If programmers are told to avoid writing **goto**s whenever possible, the flow of program control is kept simple; similarly if programmers are told to avoid writing "suspicious" programs, the flow of data (i.e. the patterns of assignment and evaluation of variables) is also kept simple.

Programs are easier to test and debug because an error is more likely to be detected immediately and not at some future point in the program. It is even worse when the absence of checks cause no fault to be reported except giving results that are known to be wrong; this case is the hardest

for programmers to put right, the error causing the problem might be anywhere in the program.

Fewer computer runs are required (a side effect of easier debugging).

On the other hand, the costs are more obvious:

Extra CPU, each statement takes longer to execute

Extra memory, each variable requires more space.

CPU and memory are cheap and still getting cheaper. We remember that the same arguments were raised by Fortran programmers in the 1960s who refused to admit the desirability of array subscript checks. The question we must ask is not "Can we afford these costs" but "Can we risk not making these checks?"

7.6 Certificates of airworthiness

(Based on information supplied by the Airworthiness Division of the Civil Aviation Authority)

There are many other industries producing complex structures which must be proved to have acceptable performance and safety. Civil aircraft must be granted a Certificate of Airworthiness (C of A). This appendix is a brief description of a process which in practice is a very complex business, more especially so when one considers the various nuances on the subject that exist between the certification authorities and aircraft constructors around the world.

To comply with British Law as contained in the Air Navigation Order, the Civil Aviation Authority (CAA) must be satisfied, before a C of A is issued, that an aircraft is fit to fly having regard to its design, manufacture, workmanship and materials, and to the results of any flying tests. The Air Navigation Order does not specify a code of airworthiness requirements by which "fit to fly" may be judged. However, for a C of A to be issued which can claim international acceptance for flights over other countries, the Authority must satisfy itself that the aircraft complies with the "detailed and comprehensive" national code which the UK has lodged with International Civil Aviation Organization (ICAO).

The UK's detailed and comprehensive code, British Civil Airworthiness Requirements (BCAR), has been developed over the years by the Authority in conjunction with the manufacturing and operating industry in the UK (taking account also of foreign airworthiness codes) and provides a detailed basis on which the airworthiness of an aircraft may be assessed. More recently a joint European code known as JAR 25 has been developed within Europe and has been adopted by UK and some other European countries as their national code. While many requirements are simple and quantitative, a very considerable number require qualitative judgements and in many cases interpretation of the generalized intent of a requirement in relation to specific circumstances is necessary. In the UK the Authority discharges its task by investigating the aircraft constructor in sufficient depth so that it can be satisfied that proper procedures have been established by the constructor

to make all the checks and balances necessary in such a complex task as the design, manufacture and testing of an aircraft. The constructor's procedures for manufacturing processes, quality control, and testing will all be examined and monitored on a continuing basis.

When the constructor embarks on the design of a new aircraft, the Authority is brought in at a very early stage. A CAA design liaison surveyor will be appointed to co-ordinate the Authority's investigation of the design and the airworthiness standards applicable in the particular case will be notified to the constructor.

As the design proceeds, CAA specialists visit the design organization to discuss the methods whereby compliance with the airworthiness standards will be established, to witness such tests as they consider necessary, to review the results of tests and to participate in the flying trials as necessary to determine compliance with the requirements.

An inspection surveyor is nominated to monitor the progress of manufacture, and he makes sampling checks on the quality control achieved during construction.

When the aircraft reaches the stage of having completed all necessary ground and flight tests, the constructor will certify to the Authority in writing that the aircraft complies with the appropriate airworthiness requirements.

At this stage the CAA will make a report on its investigation to the Airworthiness Requirements Board (ARB) which is an independent body representative of aviation interests. The Authority is required by the Civil Aviation Act 1971 to seek the advice of the ARB on the standards by reference to which a C of A may be granted, and whether or not a new type of aircraft complies with the standards. After receiving the advice of the ARB, the CAA will decide whether or not to issue a Type Certificate and subsequently a C of A for aircraft of the type. (Note: if the CAA decides not to take the advice of ARB it must publish its reasons publicly).

Questions and answers

1. Are tests recognized internationally or does each country have its own requirements which must be satisfied? If a universal test is sufficient, who specifies and performs it?

   CAA certification of a UK constructed aircraft is to the UK detailed and comprehensive national code lodged with ICAO, and this is acceptable to all signatory countries for flight over or landing in their country of a UK registered aircraft. Many countries have their own code of requirements (or use the American code, FAR) and in this case they *may* require further investigation of a UK aircraft before it can be granted a C of A on that country's register.

   A recent development is the agreement in Europe on the Joint Airworthiness Requirements for large aeroplanes, engines, etc. which is a common code acceptable by 10 European countries.

2. Economics. How is the testing scheme financed? If the manufacturer pays, what is the carrot (stick?) which ensures aircraft are tested? Who develops the tests, and who pays for their specification?

In the UK the work by the CAA Airworthiness Division has to be paid for by the "Applicant" (usually the aircraft constructor). Test work required by the CAA which the constructor has to perform is also at his cost. The incentive is simple – unless the work is done, no C of A is granted and without that the aircraft is not permitted to fly (except, of course, for test purposes).

The airworthiness tests, and other airworthiness requirements, are developed by the CAA in consultation with the aircraft industry.

The matter of finance varies around the world. In some cases, e.g. the USA, the Airworthiness Authority's charges are in effect borne by central government. The manufacturer still has to fund his own testing.

3. Is testing a continuous process as an aircraft is designed and built, or is it done only after the prototype is complete?

During the design and development of an aircraft typically thousands of tests are made. Some of these will be on wind tunnel models to determine the physical shape of the aircraft, some will be on structural details such as skin joints etc. to determine the most efficient design, and many will be on components and equipment of the aircraft to determine their performance. While many of these tests will not be primarily for certification purposes, many will make a direct contribution to establishing compliance with the requirements.

Where the design has been finalized it is usual for an almost complete test airframe to be constructed and subjected to a range of ultimate and fatigue tests for certification purposes. This test work often continues on well beyond the date of certification in order to provide data on the behaviour of the structure ahead of the experience accumulated in service.

Flight testing is made during the aircraft's development by the constructor to investigate characteristics throughout the flight envelope, to resolve problems, to measure performance and to establish compliance with the flight requirements. The CAA participates in the latter and makes whatever qualitative judgements may be necessary.

4. Must tests be repeated at intervals or is testing performed only once? What happens when an aircraft is modified (e.g. stretched, different engines, a freight version), who decides whether fresh tests are required or old tests must be repeated?

After certification of the initial aircraft of a type, all production aircraft are subjected to a less extensive flight test programme by the constructor, with CAA participation, to check that production aircraft behave in the same way as the prototype. In addition all aircraft on the UK register are flight tested periodically throughout their service lives to check for signs of deterioration in their flying qualities or performance.

If the modifications to an aircraft have significant effect on its structure,

aerodynamics or power plants then additional airworthiness investigation and testing is necessary. The extent of the testing will be decided by the CAA in relation to the significance of the changes.

5. Is an aircraft judged only as a complete aircraft or is the manufacturer required to prove that the method of design and construction is satisfactory?

As you will see from the above, the aircraft is investigated in great depth not only part by part, system by system, but also as a total machine. The constructor is required to demonstrate that his competence in the various aspects of design and construction is adequate. In some specific areas where CAA may have doubt about a constructor's expertise, we may require that expert assistance in this field be obtained. This would, of course, be unusual in relation to the large constructors, but may well occur when a small firm is breaking new ground.

6. Is each part of an aircraft judged separately? It might be that each individual part is satisfactory but that two parts interact together in a dangerous manner.

See 3 and 5 above. With complex aircraft, we require that systems safety assessments be made for each individual system or sub system to ensure a rigorous and disciplined examination of the possible failures and their effects. These then have to be combined into a global analysis to consider the effect on the aircraft as a whole. The possibility of dangerous interaction, or dangerous increases in crew workload are examined.

7. Is there any scheme for reporting faults discovered after the tests have been made and requiring them to be put right?

All UK constructors are required to keep the CAA informed of defects and failures which might undermine the basis of certification as they are discovered, whether they occur on the constructor's own aircraft, a UK registered aircraft or a foreign registered aircraft. While CAA has no jurisdiction over foreign registered aircraft, and cannot compel foreign operators of UK constructed aircraft to report defects and incidents, the Air Navigation Order requires that all significant defects and incidents occurring on UK registered aircraft in service be reported to the CAA under the Mandatory Occurrence Reporting system. CAA examines all such occurrences and investigates those with significant airworthiness implications to ensure that corrective action, where necessary, is taken. Weekly listings of the incidents reported are circulated to all operators for their information and regular summary reports are published by the CAA. The data is examined at intervals to detect trends which might be developing.

7.7 MOD Contractor Assessment for Computer Software

The accreditation system used by the Ministry of Defence examines the method whereby software is produced rather than actual software. The process of conducting an assessment is defined in a manual [MOD 78a] and the matters covered by the assessment are defined in another manual [MOD 78b]. The assessment looks at many facets of the way software is produced and managed, in particular it ensures that:

(a) There is a good software management organization and, there is a manager responsible for software quality assurance who has sufficient authority and independence to resolve problems.

(b) Adequate plans are made for all parts of software projects, i.e. design, development, testing and maintenance. Project milestones are identified so that progress may be monitored.

(c) Standards are laid down and enforced for documentation, programming, and testing. There are periodic design reviews to monitor progress, record agreed changes, and ensure the achievement of performance, reliability and maintainability. There are procedures to detect discrepancies affecting quality, and to take corrective action.

(d) Methods of testing and inspection are effective and proper records are kept.

(e) The company requires that software purchased from outside is itself inspected and tested to ensure that it satisfies the same high standards.

(f) The company periodically reviews its quality control procedures and takes steps to correct any deficiencies.

(g) There are effective procedures that ensure that copies of software are adequately identified and checked to be correct.

8. Acknowledgements

The drafts of this report have been read by a number of colleagues and experts in compiler validation. Although any remaining errors are of course our responsibility, we are grateful for all their suggestions and contributions. Among those who commented are: A M Addyman, G N Baird, J G P Barnes, F M Blake, P R Brown, J W Charter, R J Cichelli, J V Cugini, M H Forrester, J B Goodenough, G Goos, D Grune, A J Heath, H Huenke, N Malagardis, A H J Sale, T D Wells, B A Wichmann.

9. References

[ANSI77] American National Standards Institute, Inc,
Proposed American national standard programming language for Minimal Basic,
ANSI X3.60-1977, American National Standards Institute, 1430 Broadway, New York, NY 10018, USA.

[ANSI78] American National Standards Institute, Inc,
American national standard programming language Fortran,
ANSI X3.9-1978, American National Standards Institute, 1430 Broadway, New York, NY 10018, USA.
The language commonly known as Fortran 77.

[BAIL77] C B Bailey, R E Jones,
Accuracy of CDC 6600/7600 Fortran library functions,
SAND77-1038, Oct 1977, Sandia Laboratories, Albuquerque, New Mexico,

USA, 87115.

[BAIR79] G N Baird,
Compiler validation from a functional point of view,
In [NCC 80], pp 29-43.

[BRAU80] W Brauer (Editor),
Net theory and applications,
Lecture notes in computer science 84, Springer Verlag, 1980.

[BJOR80] D Bjorner (Editor),
Abstract software specifications (1979 Copenhagen Winter School Proceedings),
Lecture notes in computer science 86, Springer Verlag, 1980.

[BROW77] W S Brown,
A realistic model of floating-point computation,
Mathematical software III, (edited by J R Rice), Academic Press, pp343-360, 1977.

[BROW80a] W S Brown, S I Feldman,
Environmental parameters and basic functions for floating-point computation,
Computer science technical report No 72, March 1980,
Bell Labs, New Jersey 07974, USA.

[BROW80b] W S Brown,
A simple but realistic model of floating-point computation,
Computer science technical report No 83, May 1980,
Bell Labs, New Jersey 07974, USA.

[BSI 79] British Standards Institution,
Draft Standard Specification for the computer programming language Pascal,
Document 79/60528DC, Mar 1979.

[BSI 80] British Standards Institution,
Specification for computer programming language RTL/2,
British Standard BS5904:1980, 1980.

[CODY80] W J Cody, W M Waite,
Software manual for the elementary functions,
1980, Prentice Hall, Englewood Cliffs, New Jersey, USA.

[CUGI80a] J V Cugini,
Testing and the Basic software standard, 1980,
Institute for Computer Sciences and Technology, National Bureau of Standards, Washington, DC, USA, 20234.

[CUGI80b] J V Cugini, J S Bowden, M W Skall,
NBS Minimal Basic test programs - Version 2 - User's manual,
Volume 1 - Documentation, Volume 2 - Source listings and sample output,
November 1980,
NBS Special Publication 500-70/1 and 500-70/2,
Institute for Computer Sciences and Technology, National Bureau of

Standards, Washington, DC, USA, 20234.

[DAVI79] A M Davis, T G Rauscher,
Formal techniques and automatic processing to ensure correctness in requirements specifications
In [IEEE79], pp15-35.

[DEKK71] T J Dekker,
A floating-point technique for extending the available precision,
Numer Math, Vol 18, pp224-242, 1971.

[DEMO76] R M De Morgan, I D Hill, B A Wichmann,
Modified report on the algorithmic language ALGOL 60,
Computer Journal, Vol 19, pp364-379, 1976.

[FCTC78] Federal Compiler Testing Center (CFT),
Fortran compiler validation system - FCVS78 - user's guide,
Version 1.0, Nov 1978,
General Services Administration, Two Skyline Place, Suite 1100, 5203 Leesburg Pike, Falls Church, VIRGINIA, USA, VA 22041.

[FOST80] K A Foster,
Error sensitive test case analysis (ESTCA9),
IEEE Transactions on Software Engineering, Vol SE-6, No 3, May 1980, pp258-264.

[GANN77] C Gannons, N B Brooks, R J Urban,
JAVS technical report, user's guide,
Rome Air development center, New York, Report RADC-TR-77-126, Volume 1. (USGR Ref No AD A040 103, Volume 2 is AD A040 104, A final report is AD A041 237).

[GARW66] J V Garwick,
The definition of programming languages by their compilers,
In - Formal language description languages for computer programming, (Edited by T B Steel), North Holland Publishing Company, 1966, pp139-147.

[GILS78] D E Gilsinn, C L Sheppard,
NBS Minimal Basic test programs - Version 1,
(four volumes) NBSIR 78-1420-1 to 4, 1978, Institute for Computer Sciences and Technology, National Bureau of Standards, Washington, DC, USA, 20234.
Volume 1 introduces the test suite and gives a general overview, volumes 2, 3 and 4 contain brief descriptions, listings and sample outputs of the individual test programs.

[GOOD75] J B Goodenough, S L Gerhart,
Toward a theory of test data selection,
IEEE Transactions on Software Engineering, Vol SE-1, No 2, June 1975, pp156-173.

[GOOD80] J B Goodenough,
The Ada compiler validation capability,
SIGPLAN symposium on the Ada programming language, Boston, Massachusetts, 9-11 Dec 1980.

[GRUN79] D Grune,
The revised MC Algol 68 test set,
IW 122/79, November 1979, Department of Computer Science, Mathematisch Centrum, Amsterdam, The Netherlands.

[HOLB74] F E Holberton, E G Parker,
NBS Fortran test programs (Vol 1 Documentation for versions 1 and 3; Vol 2, Listings for version 1; Vol 3, Listings for version 3),
NBS Special publication 399, October 1974, Institute for Computer Sciences and Technology, National Bureau of Standards, Washington, DC, USA, 20234.

[HOYT77] P M Hoyt,
The Navy Fortran validation system,
FCCTS/TR-77/18, May 1977, ADPE Selection Office, Department of the Navy, Washington, DC, USA 20736. (USGR Ref No AD A039 770).

[ICHB79a] J Ichbiah, et al,
Preliminary Ada reference manual,
SIGPLAN Notices, Vol 14, No 6, June 1979, Part A.

[ICHB79b] J Ichbiah, J G P Barnes, J C Hellard, B Krieg-Brueckner, O Roubine, B A Wichmann,
Rationale for the design of the Ada programmming language,
SIGPLAN Notices, Vol 14, No 6, June 1979, Part B.

[ICHB80] J Ichbiah, et al,
Reference manual for the Ada programming language,
July 1980, Honeywell Inc, Systems and Research Center, 2600 Ridgway Parkway, Minneapolis, MN 55413, USA.

[IEEE79] IEEE,
Proceedings of Conference Specifications of Reliable Software,
April 1979, IEEE Catalog No. 79 CH1401-9C (British Library, Lending Division, Ref 79/14863).

[ISO 79] International Organization for Standardization,
Second draft proposal - Minimal Basic, ISO/DP 6373,
International Organization for Standardization, DP 6373 (ISO/TC 97/SC 5/N 456), Jan 1979.

[ISO 80] International Organization for Standardization,
Draft standard specification for the programming language Pascal, DP 7185,
International Organization for Standardization, DP 7185 (ISO/TC 97/SC 5/N 565), Feb 1980.

[JONE79] C Jones,
A survey of programming design and specification techniques,
In [IEEE79], pp 91-103.

[MALC72] M A Malcolm,
Algorithms to reveal properties of floating-point arithmetic,
Comm ACM, Vol 15, pp 949-951, 1972.

[MOD 78a] Ministry of Defence,

Guide to contractor assessment. Book 1. The conduct of contractor assessment. 1978.
Director General of Quality Assurance MOD(PE).

[MOD 78b] Ministry of Defence.
Guide to contractor assessment. Book 4. Computer software QA systems. 1978.
Director General of Quality Assurance MOD(PE).

[NCC 80] National Computing Centre.
Language implementation validation.
Proceedings of the two day workshop held at Manchester on 12-13 September 1979. edited by D J Dwyer and D I Noble.
National Computing Centre. Manchester.

[NTIS79] United States National Technical Information Service.
AD A036 174 CCVS74 V3.0 User's Guide .
National Technical Information Service. 5285 Port Royal Road. Springfield. Virginia 22151. USA.

[OSTE76] L J Osterweil. L D Fosdick.
DAVE - a validation error detection and documentation system for Fortran programs.
Software practice and experience. Vol 6. pp473-486. 1976.

[PETE77] J L Peterson.
Petri nets.
Computing Surveys. Vol 9. pp 223-252. 1977.

[PETR62] C A Petri.
Kommunikation mit automaten.
Schriften des Rheinisch-Westfalieschen Institues fur Instrumentelle Mathematik an der Universitat Bonn. Heft 2. Bonn. W Germany. 1962.

[REIM78] G W Reimheer.
Computer Software Standards. A bibliography with abstracts.
NTIS report NTIS/PS-78/0541. June 1978.
National Technical Information Service. Springfield. Virginia. VA 22161.

[REIN79] C H Reinsch.
Principles and preferences for computer arithmetic.
SIGNUM Newsletter. pp12-27. March 1979.

[ROBI73] R A Robinson. D R Williams.
Jovial compiler validation system user's guide - volume 1.
Rome Air development center. Griffiss Air Force Base. New York. 13441. (USGR Ref No AD - 772 747).

[ROUB76] O Roubine.
The design and use of specification languages.
Stanford Research Institute. Technical Report CSD 48. Oct 1976 (AD A038 783).

[SALE79] A H J Sale. R A Freak.
Four sample validation reports.

Pascal News, No 16, pp142-153, dated October 1979.

[SCOW79] R S Scowen,
A new technique for improving the quality of computer programs,
Proceedings 4th IEEE International conference on software engineering,
Technical University, Munich, Sept 17-19, 1979, pp73-78.

[SIGN79] SIGNUM Newsletter,
The proposed IEEE floating-point standard,
ACM SIGNUM newsletter, special number, October 1979, pp1-32.

[SOFT80a] Softech (J B Goodenough, J R Kelly),
Ada compiler validation capability - Long range plan,
Softech report 1067-1.1, Feb 1980,
Softech Inc, 460 Totten Pond Road, Waltham, MA 02154, USA.

[SOFT80b] Softech,
Preliminary Ada compiler validation implementers' guide,
Softech report 1067-2, April 1980,
Softech Inc, 460 Totten Pond Road, Waltham, MA 02154, USA.

[SOFT80c] Softech (J B Goodenough, J R Kelly, N Lomuto, R Mandl, B A Wichmann),
Ada compiler validation implementers' guide,
Softech report 1067-2.3, October 1980,
Softech Inc, 460 Totten Pond Road, Waltham, MA 02154, USA.

[STRA65] C Strachey,
An impossible program,
Computer Journal, Vol 7, pg313, 1965.

[WARS62] S Warshall,
A theorem on Boolean matrices,
Journal ACM, Vol 9, pp 11-12, 1962.

[WEYU80] E J Weyuker, T J Ostrand,
Theories of program testing and the application of revealing subdomains,
IEEE Transactions on Software Engineering, Vol SE-6, No 3, May 1980,
pp236-246.

[WICH73] B A Wichmann,
Some validation tests for an ALGOL 60 compiler,
National Physical Laboratory, Report NAC33, 1973.

[WICH76] B A Wichmann, B Jones,
Testing Algol 60 compilers,
Software practice and experience, Vol 6, pp261-270, 1976.

[WICH79] B A Wichmann, A H J Sale,
A Pascal processor validation suite - version 2.2,
Pascal News, No 16, pp10-142, dated October 1979.

Appendix A
Extracts from the Pascal Test Suite

In this Appendix we give several samples of the test programs from the current suite. The purpose of this is to illustrate the nature of the suite so that the reader does not have to consult the bulky computer listings of the complete suite. The samples are chosen to be representative of the whole suite in general, although for specific questions on testing individual language features the complete listing would have to be consulted.

Conformance Tests

These are always correct Standard conforming Pascal programs. They can nevertheless fail on a correct implementation if the basic assumptions are not met (see Appendix C). They vary from trivial tests on the lexical and syntactic issues of the language to very obscure semantic tests. The general philosophy behind all the tests is to make them as demanding as possible. (This should be contrasted with the COBOL tests.)

```
 1 {TEST 6.1.9-1, CLASS=CONFORMANCE}
 2
 3 {: This program checks that the two equivalent forms of comment
 4    delimiters are implemented correctly. }
 5 {  It contains four comments with all the permutations of delimiters.
 6    Processors are not allowed to ignore one form of comment delimiter
 7    unless they do not have the appropriate characters in their set. }
 8 {V3.0: Test revised in line with new treatment in DP7185, and moved
 9    to appropriate section (was test 6.1.8-3). }
10
11 program t6p1p9d1(output);
12 var
13    i : 0..4;
14 begin
15    i := 0;
16    { This is a standard comment }
17    i := i+1;
18    (* This is an alternative form *)
19    i := i+1;
20    { This, though correct, is a misleading practice *)
21    i := i+1;
22    (* These equivalences allow for greater portability }
23    i := i+1;
24    if (i=4) then
25       writeln(' PASS...6.1.9-1')
26    else
27       writeln(' FAIL...6.1.9-1')
28 end.
```

Lines 3-4 contains the comment which will be printed out by the automatic analysis package if the test fails (i.e. does not print 'PASS...6.1.9-1'). Lines 8-9 contains the comment which indicates the history of the test since the last

change to it. For the previous history, the earlier versions of the suite must be consulted. Lines 5–7 contain the extended comment explaining how the general objective of the test has been achieved. Occasionally, this comment can be quite long and detailed if the test itself is convoluted and not self-documenting. The test itself is very straightforward in this case. However, if a processor skipped from "(" to the matching ")", then one statement would be skipped and the test would fail. Most tests are written with some idea as to how they can fail (sometimes they are based upon actual failures). In general, there is no attempt to diagnose the cause of the error which could perhaps be aided by printing out i. The tests are such that if they do fail, then a compiler-writer should be able to diagnose the reason very quickly. Lines 25 and 27 print the highly stylized output. This form of output is analysed by the report generation package.

```
 1 {TEST 6.6.3.3-3, CLASS=CONFORMANCE}
 2
 3 {: This test checks that if a variable passed as a parameter
 4    involves the indexing of an array, or the dereferencing of a
 5    pointer, then these actions are executed before the activation
 6    of the block. }
 7 {V3.0: Rewritten to include type rekptr = ↑rekord
 8    Write for FAIL elaborated. }
 9
10 program t6p6p3p3d3(output);
11 type
12    rekptr = ↑rekord;
13    rekord = record
14                a : integer;
15                link : rekptr;
16                back : rekptr
17             end;
18 var
19    urray : array[1..2] of integer;
20    i     : integer;
21    temptr,ptr : rekptr;
22 procedure call(arrayloctn : integer;
23                ptrderef : integer);
24    begin
25       i:=i+1;
26       ptr:=ptr↑.link;
27       if (urray[i-1] <> arrayloctn) or
28          (ptr↑.back↑.a <> ptrderef) then
29          writeln(' FAIL...6.6.3.3-3')
30       else
31          writeln(' PASS...6.6.3.3-3')
32    end;
33 begin
34    urray[1]:=1;
35    urray[2]:=2;
36    i:=1;
37    new(ptr);
38    ptr↑.a:=1;
39    new(temptr);
40    temptr↑.a:=2;
41    ptr↑.link:=temptr;
42    temptr↑.back:=ptr;
43    call(urray[i],ptr↑.a)
```

```
44 end
```

This program is typical of the more convoluted semantic tests. A problem with the definition of all programming languages is that of the order of operations. Pascal defines the order of some operations but leaves others as being *implementation-dependent* (see the relevant test below). In this case, we are testing that the parameters to a procedure call have been completely evaluated before the body is executed. To test this, we must arrange that the evaluation of the parameters interacts with the statements of the body so that different effects would be obtained. These sort of tests are rarely a problem with Pascal compilers because few do significant optimization. However, if the procedure call was expanded in-line, then the dangers can be more easily appreciated. In this case, a possible error could be caused by the change in the value to i and ptr so that the checks in the procedure would fail. Note that if the addressing code produced by the compiler is incorrect so that the program crashes, then "PASS...6.6.3.3-3" will not be printed and the automatic analysis system will report a failure.

Deviance tests

These are the complimentary tests to the conformance ones since this class consists of incorrect programs. They should not execute at all, even if the incorrect construct is in a part of the program which would not be executed. In a similar manner to the conformance tests, there is a wide variation from simple lexical and syntactic issues to programs in which the error is far from obvious. It is difficult to write good deviance tests because to be useful they need to be *nearly* correct in a manner that could potentially confuse a compiler.

```
 1 {TEST 6.2.1-10, CLASS=DEVIANCE}
 2
 3 {: A processor should not allow multiple const and type parts
 4    before a var part. }
 5 {V3.0: New test in 3.0 to detect multiple const and type
 6    parts in declarations. }
 7
 8 program t6p2p1d10(output);
 9
10 const nullstring='        ';
11 type  string8   =packed array[1..8] of char;
12
13 const linelength=80;
14 type
15    lineindex=1..80;
16    line=array[lineindex] of char;
17
18 var
19    l:line; s:string8; i:lineindex;
20
21 begin
22    s:=nullstring;
23    for i:=1 to linelength do l[i]:=' ';
24    writeln(' DEVIATES...6.2.1-10')
25 end.
```

Many people have observed that the restriction in Pascal on the order of

declarations can hinder program modularity. Moreover, even with the one pass restriction, it is easy to extend a Pascal compiler to accept multiple constant definition parts etc. Hence the restriction within the language is merely so that one knows where everything is. Although many compilers have been extended so that this test will be accepted, Jensen and Wirth and the Standard are quite clear in rejecting this.

A processor which permits this type of extension should provide an optional entry to the compiler where only Standard-conforming programs are accepted. In this way a compiler can be constructed which is compatible with the existing extension and yet capable of being validated. The "Standard only" switch must not be invoked by a comment since no textual changes are permitted during a formal validation.

```
 1 {TEST 6.4.5-13, CLASS=DEVIANCE}
 2
 3 {: This test checks that structurally identical array-types which are
 4    even textually identical in their declarations are not considered
 5    identical. }
 6 {  The test is similar to 6.4.5-8, but is repeated to ensure that
 7    idiosyncratic behaviour for arrays of char does not mask effects
 8    for arrays of other component types. }
 9 {V3.1: Comment changed. }
10
11 program t6p4p5d13(output);
12
13 type
14    index = 1..10;
15    urrayone = array[index] of boolean;
16    urraytwo = array[index] of boolean;
17 var
18    arraytwo : urraytwo;
19
20 procedure test(var urray : urrayone);
21 begin
22    writeln(' DEVIATES...6.4.5-13')
23 end;
24
25 begin
26    { The two types urrayone and urraytwo are not identical,
27      and thus the call to test should not be acceptable. }
28    test(arraytwo)
29 end.
```

Type compatibility is a well-known trouble-spot in Pascal. Most of the P-code style of compilers will accept this test even though it is clearly illegal according to the Standard. Of course, type compatibility influences many parts of the language, but parameter passing is critical. With **var** parameters as in this example, the information passed is often just an address and hence there is no logical difficulty in giving a meaning to this test. Note that between lines 27 and 28 one could insert **if false then** so that the call of the procedure is not executed. But the program would still be illegal.

Error-handling tests

As explained in Chapter 4, the current suite includes a pretest with every error-handling test. So here we give an example of each for the same error. These tests are not as easy to interpret as the conformance or deviance tests. Firstly, there is no *requirement* that any error should be detected. The Standard merely hints that the early detection of errors is highly desirable. Secondly, if an error is detected, in quite a few circumstances this could be during compilation rather than the more typical case of during execution. Thirdly, the actual error situations vary from ones which are simple to detect, to those which no current implementation is known to detect. Lastly, an implementation has to state *which* errors are always detected. The Standard conveniently lists and numbers the 59 error situations in an appendix (strictly speaking this appendix is not part of the Standard).

For the automatic analysis of the test results, the detected and undetected error situations must be checked against the claims made by the compliance statement of the processor. To aid this process, the reference number of the situation appears in the header comment (both for the test and pretest). The error is not detected if both the test and pretest execute to completion. The error is detected if the pretest executes to completion and the test does not. If the pretest does not execute to completion, then a failure has occurred, that is, the pretests are regarded as an additional conformance test.

```
 1 {PRETEST 6.5.5-2, CLASS=ERRORHANDLING, NUMBER= 6}
 2
 3 program p6p5p5d2(output);
 4 var
 5    fyle : text;
 6 procedure naughty(var f : char);
 7    begin
 8       if f='H' then
 9          ;
10    end;
11 begin
12    rewrite(fyle);
13    fyle↑:='H';
14    naughty(fyle↑);
15    writeln(' PRETEST...6.5.5-2')
16 end.
```

```
 1 {TEST 6.5.5-2, CLASS=ERRORHANDLING, NUMBER= 6}
 2
 3 {: This program causes an error to occur by changing the
 4    current file position of a file, while the buffer
 5    variable is an actual variable parameter to a procedure. }
 6 {  The error should be detected by the processor. }
 7 {V3.1: Comment changed. }
 8
 9 program t6p5p5d2(output);
10 var
11    fyle : text;
12 procedure naughty(var f : char);
13    begin
14       if f='G' then
15          put(fyle)
```

```
16     end;
17 begin
18     rewrite(fyle);
19     fyle↑:='G';
20     naughty(fyle↑);
21     writeln(' ERROR...6.5.5-2');
22     writeln(' ERROR NOT DETECTED')
23 end.
```

Many of the error situations listed in the Standard are on the mis-use of files. The majority of these are very simple to detect which makes it surprising that the detection is not a requirement of the Standard. However, many involve a complex interaction of the types used and the dynamic behaviour of the program (perhaps requiring the data for the program to resolve the legality of the construct). The example illustrated here is one such complex interaction involving file operations.

The file operations alter the state of the file and its associated file buffer. In other respects, the file buffer is just another variable which means that it can be assigned to, be passed as a var parameter etc. This creates a conflict of use if a file operation is performed when the file buffer can be accessed by another route. This is prohibited by the Standard. Ideally, the prohibition should be a static check but unfortunately there is no simple way to formulate such a restriction. Hence the Standard makes it an error. The error is on line 15 of the test when a file operation is performed on `fyle`. The error is not evident from the procedure `naughty` itself since it is the call that forces the error. In this case, because of the call on line 20, the procedure `naughty` performs a file operation while an access exists to the file buffer via the var parameter `f`.

In this case it is easy to design a corresponding pretest which is legal Pascal by just omitting the file operation. Note that although static detection is possible in this case, it can depend upon the data if the file operation is involved only when certain dynamic conditions arise. In fact, an alternative method of producing the pretest would be to leave the file operation in, but change line 14 to be **`if f='G' then.`**

Quality tests

These tests do not formally belong to the validation suite in the strictest sense. For instance, a failure of any of these tests would not inhibit formal validation. The purpose of them is to probe the limits of an implementation in ways which cannot be justified from the Standard alone. As an example, the Standard explicitly states that questions about the capacity of a processor are not addressed. On the other hand, if an implementation does not permit procedures to be nested more than two deep or permit more than 100 bytes of constants, then the processor cannot be regarded as being of high quality. In fact, the existing quality tests are in four classes:

1. Performance tests. These are much more like user programs than the rest of the suite. They can be used to estimate the speed of execution of Pascal.

2. Floating point tests. These are largely a recoding of some FORTRAN tests of Cody and Waite (see Chapter 4). They are not illustrated here because of their size and that they are

already available in Cody and Waite's book.

3. Linear complexity tests. Pascal allows many syntactic items to be repeated in simple lists. These programs test for a large limit on one such list, as illustrated below.

4. Nested complexity tests. In common with all Algol-like languages, Pascal provides many nested constructs. These programs test for a large limit to one such nesting, as illustrated below.

```
1 {TEST 6.1.7-14, CLASS=QUALITY}
2
3 {: This program checks that processors allow a reasonably large
4    (20) number of constant strings in a program. }
5 {V3.0: New test. }
6
7 program t6p1p7d14(output);
8 const
9    s1 = 'STRING 01: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
10   s2 = 'STRING 02: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
11   s3 = 'STRING 03: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
12   s4 = 'STRING 04: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
13   s5 = 'STRING 05: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
14   s6 = 'STRING 06: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
15   s7 = 'STRING 07: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
16   s8 = 'STRING 08: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
17   s9 = 'STRING 09: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
18   s10= 'STRING 10: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
19   s11= 'STRING 11: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
20   s12= 'STRING 12: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
21   s13= 'STRING 13: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
22   s14= 'STRING 14: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
23   s15= 'STRING 15: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
24   s16= 'STRING 16: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
25   s17= 'STRING 17: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
26   s18= 'STRING 18: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
27   s19= 'STRING 19: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
28   s20= 'STRING 20: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ';
29 begin
30 if (s1 = s2) or (s3 = s4) or (s5 = s6)
31    or (s7 = s8) or (s9 = s10)
32    or (s11 = s12) or (s13 = s14) or (s15 = s16)
33    or (s17 = s18) or (s19 = s20) then
34    writeln(' FAIL...6.1.7-14')
35 else
36    writeln(' QUALITY...6.1.7-14')
37 end.
```

A feature of many poor quality implementations of Pascal is a low limit on various features, especially those related to code generation. The above test examines the size of the pool for string constants (at least 940 characters). In order to be confident that the space has indeed been allocated, lines 30-33 perform a crude check on the values. One can claim that the test is defective in that an optimizing compiler could reduce the program to the single write statement on line 36. Such problems are inevitable with the suite, but even in this case it

would serve to check the optimization. The use of the colon in the strings is perhaps not advisable in that colon is not required for the character set of the target (i.e, it is needed to represent Pascal source programs but not for their execution). However, this is one of the basic assumptions, see Appendix C.

```
 1 {TEST 6.8.3.9-20, CLASS=QUALITY}
 2
 3 {: This program checks that for loops can be nested
 4    15 deep. }
 5 {V3.0: Value check added.
 6    Extended nesting depth from 12 to 15. }
 7
 8 program t6p8p3p9d20(output);
 9 var a0, a1, a2, a3, a4, a5, a6,
10    a7, a8, a9, a10, a11, a12, a13, a14: integer;
11    sum: integer;
12 begin
13    sum := 0;
14    for a0 := 1 to 2 do
15       begin
16       sum := sum + 1;
17       for a1 := 1 to 2 do
18          begin
19          sum := sum + 1;
20          for a2 := 1 to 2 do
21             begin
22             sum := sum + 1;
23             for a3 := 1 to 2 do
24                begin
25                sum := sum + 1;
26                for a4 := 1 to 2 do
27                   begin
28                   sum := sum + 1;
29                   for a5 := 1 to 2 do
30                      begin
31                      sum := sum + 1;
32                      for a6 := 1 to 2 do
33                         begin
34                         sum := sum + 1;
35                         for a7 := 1 to 2 do
36                            begin
37                            sum := sum + 1;
38                            for a8 := 1 to 2 do
39                               begin
40                               sum := sum + 1;
41                               for a9 := 1 to 2 do
42                                  begin
43                                  sum := sum + 1;
44                                  for a10 := 1 to 2 do
45                                     begin
46                                     sum := sum + 1;
47                                     for a11 := 1 to 2 do
48                                        begin
49                                        sum := sum + 1;
50                                        for a12 := 1 to 2 do
51                                           begin
```

```
52                                                  sum := sum + 1;
53                                                  for a13 := 1 to 2 do
54                                                     begin
55                                                     sum := sum + 1;
56                                                     for a14 := 1 to 2 do
57                                                        begin
58                                                        sum := sum - 1
59                                                        end
60                                                     end
61                                                  end
62                                               end
63                                            end
64                                         end
65                                      end
66                                   end
67                                end
68                             end
69                          end
70                       end
71                    end
72                 end
73              end;
74     if sum <> -2 then
75        writeln(' FAIL...6.8.3.9-20')
76     else
77        writeln(' QUALITY...6.8.3.9-20')
78 end.
```

The above example is that of complexity via nesting. The nesting of one construct within itself is tested rather than that of several different nested constructs. The reason for this is to limit the number of tests needed while being comprehensive in respect of the method used. Hence there are no tests of with statements nested in for loops nested in if statements etc. One major reason for testing this form of nesting is to reveal any limitations imposed by the code generation strategy. For instance, several Pascal compilers allocate a register for every static procedure nesting level. In consequence, there is a limit on the nesting level permitted (or perhaps within nested procedures fewer registers are available for expression evaluation). In this example, the nesting of for loops is tested. A minimal execution test is added in the hope of detecting the generation of incorrect code. If register allocation is made for the control variables, then clearly these should be allocated from the innermost outwards. There is no attempt in this case to measure the efficiency of the code generation for loops. In fact, there is another quality test for this which is a well-known benchmark.

Implementation-defined tests

In Chapter 4, a list is given of the implementation-defined aspects of Pascal as defined by the Standard. In most, but not all cases, tests can be written in Pascal to determine the nature of this feature. The most obvious case is the value of maxint. The features are numbered for easy reference in a manner similar to that for errors. The features for which no simple test can be produced are numbered 1 (subset of reals corresponding to signed-real), 5 (the point at which file operations are performed), 7 (accuracy of real operations), and 15 (the binding of file-type program parameters). Although the effect of the procedure page is implementation-defined, a test program which examines the

file would be implementation-dependent (and hence the results are not predictable).

```
 1 (TEST 6.1.9-5, CLASS=IMPLEMENTATIONDEFINED, NUMBER=16)
 2
 3 (: This program checks whether the required equivalent
 4    symbols can be used instead of the reference representation. )
 5
 6 ( The required alternative representations are for curly comment
 7   brackets and square subscript brackets. These must be provided
 8   since the necessary characters *, ), and . are available. )
 9 (V3.1 Changed to test required alternatives, not just comments)
10
11 program t6p1p9d5(output);
12 (* Test of alternate comment delimiters *)
13 var
14    x: array (. 1 .. 10 .) of boolean;
15    y: array [ 1 .. 10.) of boolean;
16 begin
17    x(.1] := true;
18    y[1.) := x[1];
19  (* test of alternate comment delimiters. If these delimiters
20    are not implemented a syntax error will result. *)
21    writeln(' OUTPUT FROM TEST...6.1.9-5' );
22    writeln(' ALTERNATE SUBSCRIPT BRACKETS IMPLEMENTED');
23    writeln(' ALTERNATE COMMENT DELIMITERS IMPLEMENTED');
24    writeln(' IMPLEMENTATION DEFINED...6.1.9-5')
25 end.
```

As indicated in the comment on lines 6-8, these alternatives must be provided. Hence the reason for including this provision under implementation-defined features. Note that if (and [brackets are not provided, then the appropriate lexical substitutions will be made, making the program inappropriate but still correct. The test can be regarded as a conformance program. The write statement on line 21 is to enable the automatic analysis package to capture the output upto line 24 so that this can appear in the test report.

Implementation-dependent tests

A list of these features is also given in Chapter 4. Two dependencies have been added which are not explicitly stated as such by the Standard. All the test programs in this class are necessarily incorrect Pascal. In consequence, no particular results can be expected and these programs would not ordinarily play any part in validation. However, the compliance statement for a processor could state that such dependencies are treated as errors and are detected. In this case, the results of the test do matter and complete execution should not happen.

The majority of the tests must use side-effects on function evaluation to determine the action taken. However, there is no guarantee that the action would be the same in other contexts. Hence one can argue that the action taken should not be printed, although this is currently done in the test suite.

Most of the tests use function calls in contexts in which function calls are rare. Hence there is a possibility that incorrect code is generated for the call so that the test crashes. This has certainly happened at least once, so that

running these tests can be useful.

```
 1 {TEST 6.7.2.3-3, CLASS=IMPLEMENTATIONDEPENDENT, NUMBER= 4}
 2
 3 {: This program determines if a boolean expression is partially
 4    or completely evaluated when the value of the expression is
 5    determined before the expression is fully evaluated. }
 6 {V3.0: Changed comment for greater clarity.
 7    Writes revised. Was previously 6.7.2.3-3 }
 8
 9 program t6p7p2p3d3(output);
10 var
11    a:boolean;
12    k,l:integer;
13
14 function sideeffect(var i:integer; b:boolean):boolean;
15 begin
16    i:=i+1;
17    sideeffect:=b
18 end;
19
20 begin
21    writeln(' OUTPUT FROM TEST...6.7.2.3-3');
22    writeln(' TEST OF SHORT CIRCUIT EVALUATION OF (A AND B)');
23    k:=0;
24    l:=0;
25    a:=sideeffect(k,false) and sideeffect(l,false);
26    if (k=0) and (l=1) then
27       writeln(' ONLY SECOND EXPRESSION EVALUATED')
28    else
29       if (k=1) and (l=0) then
30          writeln(' ONLY FIRST EXPRESSION EVALUATED')
31       else
32          if(k=1) and (l=1) then
33             writeln(' BOTH EXPRESSIONS EVALUATED')
34          else
35             writeln(' INEXPLICABLE RESULT');
36    writeln(' IMPLEMENTATION DEPENDENT...6.7.2.3-3')
37 end.
```

A well-known programming error in Pascal is to write:

```
if ptr <> nil and ptr↑.field = wanted then
```

This is not valid because an implementation is free to evaluate both arms of the boolean expression. So if the pointer is nil, then the second arm will fail. Such an error is hard to detect if the implementation only evaluates one arm.

This test illustrates this problem with the operator **and**. A similar test is also done for **or**.

Level 1 tests

These tests are for conformant arrays and have subclasses corresponding to the main classes of the suite. All the tests should be run on a processor because a level 0 system must reject all the tests. The programs themselves are rather more complex than most of the other ones because of the inherent

difficulty of adding structural type equivalence to the language. The facility naturally interacts with indexing and makes compile-time index checking more complex.

```
 1 {TEST 6.6.3.6-6, CLASS=LEVEL1, SUBCLASS=CONFORMANCE }
 2
 3 {: This test checks that the congruity of conformant array
 4    parameters is implemented correctly. }
 5 {V3.1: New test}
 6
 7 program t6p6p3p6d6(output);
 8 var
 9    fail: boolean;
10    a: array [ 1 .. 3 ] of integer;
11    b: array [ -1 .. 1 ] of array [ 0 .. 2 ] of integer;
12    i, j: integer;
13 function sum1( var a: array[ 1 .. u: integer] of integer): integer;
14    var
15       s, i: integer;
16    begin
17    s := 0;
18    for i := 1 to u do
19       s := s + a[i];
20    sum1 := s + 1 + u
21    end;
22
23 function sum2( b: array[l1..u1: integer;
24                         l2..u2: integer] of integer): integer;
25    var
26       s, i, j: integer;
27    begin
28    s := l1 + u1 + l2 + u2;
29    for i := l1 to u1 do
30    for j := l2 to u2 do
31       s := s + b[i,j];
32    sum2 := s
33    end;
34
35 procedure p( function f(var aa: array[la..ua: integer] of integer
36                         ): integer;
37               function g( bb: array[lb1..ub1: integer] of
38                                  array[lb2..ub2: integer] of integer
39                         ):integer);
40    var
41       s: integer;
42    begin
43    s := f(a) + g(b);
44    fail := s <> 129;
45    end;
46 begin
47 for i := 1 to 3 do
48    begin
49    a[i] := 10*i;
50    for j := 0 to 2 do
51       b[i-2,j] := 3*i + j;
52    end;
```

```
53 p(suml, sum2);
54 if fail then
55    writeln(' FAIL...6.6.3.6-6')
56 else
57    writeln(' PASS...6.6.3.6-6')
58 end.
```

The above test is typical of the complex language interaction between the conformant array mechanism and that of procedure parameters. Section 6.6.3.6(e) is only applicable to conformant arrays, and specifies when arrays are congruent for the static checking of formal procedure parameter calls. This can be illustrated by the example above. The call of p on line 53 is correct. This means that the actual function suml must be congruent with the formal f and similarly with the second parameter. The congruent rules must be applied again now to the parameters of f and suml. This is simple in this case since the formals on lines 13 and 35 are textually the same apart from the identifiers of the formals. However, the formals of g and sum2 must match also. In this case the congruency test works because a two-dimensional array is regarded as equivalent to an array of arrays (for conformant arrays, see 6.6.3.7).

Irregular tests

These tests can be in any class. They are irregular because they do not conform to the standards used for the rest of the suite. This means that it may not be possible to process these tests in the same way as the others. Because of this inconvenience, we have tried to minimise the number of these tests. Some, like the example below, use a non-Pascal character. Others do not have an initial comment or have a comment between the final end and the full spot. The test of the procedure page is irregular because it cannot be self-checking. The automatic analysis of these tests is not possible.

```
 1 {TEST 6.1.2-7, CLASS=DEVIANCE}
 2
 3 {: This test deviates since the character % is not a token
 4    of Pascal. }
 5 {  This test is not  relevant to processors that do not admit
 6    the % character in their character set. }
 7 {V3.1: Test moved to the 'irregular tests' file. }
 8
 9 program t6p1p2d7(output);
10 var
11    i: integer;
12 begin
13 i := 1 % 2;
14 writeln(' DEVIATES...6.1.2-7')
15 end.
```

The test is irregular because the percent character is not a valid Pascal character (i.e. is not used to represent any Pascal construct). In consequence, the source text which is provided in ISO code, cannot be converted into a character set without a percent character. This would then invalidate this test. In fact, most systems can probably handle this test in just the same way as any other (because most character sets do include the percent character).

Extension tests

This is the smallest class of tests at the moment because of the lack of any general agreement on extensions to the language. If the US Pascal group (X3J9) does agree to more extensions, then corresponding tests will be placed here. Usually, extensions are incompatible with the Standard to a minor extent. At the very least, meaning is given to an illegal program in the Standard. Hence these test would ordinarily be run in two ways, firstly with only the Standard permitted to show that it was rejected; secondly, with the extension enabled to show that the agreed interpretation was taken.

```
 1 {TEST 6.8.3.5-16, CLASS=EXTENSION, SUBCLASS=CONFORMANCE}
 2
 3 {: This test checks whether an otherwise clause in a case statement
 4    is accepted. }
 5 {  The convention is that adopted at the UCSD Pascal
 6    workshop in July 1978. The extension is accepted if the program
 7    prints EXTENSION - PASS }
 8 {V3.0: Value check made more complete. Variable 'counter' not needed.
 9    Was previously 6.8.3.5-14 }
10
11 program t6p8p3p5d16(output);
12 var
13    i,j,k:integer;
14 begin
15    j:=0; k:=0;
16    for i:=0 to 10 do
17       case i of
18       1,3,5,7,9:
19          j:=j+1
20       otherwise
21          k:=k+1
22       end;
23    if (j=5) and (k=6) then
24       writeln(' EXTENSION - PASS...6.8.3.5-16')
25    else
26       writeln(' EXTENSION - FAIL...6.8.3.5-16, OTHERWISE')
27 end.
```

This extension to Pascal is particularly useful, especially in handling unknown character sets. It is the only extension which has been agreed to date (July 1982) by the US committee. No proposals are being considered currently by the ISO Pascal working group.

Appendix B

Has the program been altered?

B. A. Wichmann

Introduction*

It is often essential to know if a computer program has been altered. If the text has been provided on magnetic tape and subsequently loaded onto another computer with possible changes in the character set, the problem becomes more difficult. This paper proposes the use of a 'parity check' as used in digital data transmission. An implementation of this algorithm in Pascal is given together with details of how a Pascal source text can be checked for alterations using the algorithm.

As explained in Chapter 5, a checking mechanism such as this is vital for third party validation. In this case, we know that Pascal will be available on the target system. Hence the provision of the checking program in Pascal solves the problem without the need to use any low-level programming or need to rely upon operating system functions. The idea is not new as can be seen from Thacher in 1962 [3]. The program also includes a number of other checks which are needed to ensure that the Pascal test suite can be easily transported.

The algorithm

The problem of the correct transmission of digital data on telephone lines has been studied in great detail. Techniques have been devised both for error detection and correction. The methods used are capable of detecting single errors and, with a very high probability, bursts of errors. In software, these would correspond to misreading a magnetic tape or the deliberate insertion or deletion of some text. Hence it appears likely that the methods used for digital transmission will be suitable for this application. The algorithm chosen is that recommended by CCITT [1]. The details are taken from the hardware implementation shown as a logic diagram in [2].

The algorithm is usually implemented in hardware attached to the transmission line. In this case, we implement it as a procedure written in Pascal. The hardware uses a 16-bit shift register which we mimic in Pascal as a boolean array (1 bit larger for program convenience):

This paper was published without the text of the Pascal program in Software – Practice and Experience, Vol 11, No 8, August 1981, pp877-879. It has been revised for this publication.


```
type
   regindex = 0 .. 16;
   ShiftRegister = array [regindex] of boolean;
```

The algorithm is a procedure which takes the next bit in the data as a parameter. A shift is applied to the register and the last value is exclusively or-ed with other elements of the register. In Pascal this becomes:

```
procedure pulse( b: boolean; var SR: ShiftRegister);
   { This algorithm follows Recommendation V41, see
     'Data Transmission Over Telephone Network: Series V
     Recommendations', International Telecommunication Union.
     Geneva (1977).
   }
   var
      i: regindex;
      e: boolean;
   begin
   for i := 15 downto 0 do
      SR[i+1] := SR[i];
   e := SR[16];
   SR[0] := b <> e;
   SR[5] := SR[5] <> e;
   SR[12] := SR[12] <> e;
   end; {pulse}
```

The final state of the shift register is the 16-bit check sum. There is one problem with this algorithm. Each call corresponds to about 300 machine instructions which is in consequence about 3000 times slower than a hardware implementation. It was felt better to use an established algorithm rather than devise another one which could be significantly faster.

The source text to be checked is conventionally regarded as a sequence of characters. Hence in this case, the algorithm is applied to each bit of the characters using separate shift registers. This gives a check sum of 16*N bits where N is the number of bits in a character.

Application to Pascal source text

We now consider the application of this algorithm to ensuring that Pascal source text has been mounted correctly on a new computer system. This works by providing the Pascal program which calculates the check digits in addition to the text to be checked. On the new system, the checking program is run with the source to be checked as data. The check digits produced are then compared by hand with those produced on the original system. No matter how large the source text to be checked, only 16*6 = 96 bits need be compared.

The checking program needs to convert a Pascal program into a standard format before applying the algorithm. The reason for this is that although (in this case) the source text is released in full ISO code, the target system need not have that capability. Hence the recipient is allowed to do the

following consistent changes to the source text.

1. Replace all alphabetic characters by a character in one case only (conventionally upper case).

2. Replace { and } by (* and *) respectively.

3. Replace [and] by (. and .) respectively.

4. Replace ↑ by @.

To allow for these changes, a table is used to convert the character sequences of the source text into a sequence of integers (of 6 bits). Each bit of the resulting integer is then subject to the procedure given above.

The conversion table is:

```
Convert: array [char] of integer;
```

Unfortunately, there is no way of initializing this array in Pascal since the values of the first and last character are not defined in the standard. This is overcome by the following:

```
const
   mincharvalue = 0;
   maxcharvalue = 255; {implementation defined value}

var
   ch:char;
begin
for ch := chr(mincharvalue) to chr(maxcharvalue) do
   Convert[ch] := 0;
end;
```

Allowance for the four characters {, }, [and] is made by always replacing them by the longer alternatives. Unfortunately, the program text:

```
if ch = '{' then
```

cannot be converted on a system which does not have { as a character. In consequence, certain sections of the Pascal checking program have to be deleted if { } or [ ] are not available.

The Pascal checking program performs a number of other functions as follows:

1. Issues a warning if a non-Pascal character is found in the source text (it is ignored in the sum check).

2. Issues a warning if, after the longer alternatives are used for the curly and square brackets, the text would exceed 72 characters width.

3. Ignores blank lines (i.e. lines containing only spaces). This is essential in practice since lines before the first or after the last are not visible and can easily be inserted (by a helpful operating system?).

4. Prints out the first non-blank line for indentification purposes.

5. Ignores initial and trailing spaces on a line. Apart from allowing an operating system to insert trailing spaces, this considerably speeds up the algorithm. Very carefully hand punched programs could also pass the check digit test (some micro implementations of Pascal only have a keyboard).

The effect of various program changes on the 96 bits is as follows:

Source Text Change	Bits different
Replace "." by "," on one line	7
Add extra line	47
Delete one line	42
One character duplicated	63
One character deleted	64
Space deleted at start of line	0
Space inserted at start of line	0
Extra blank line inserted	0
Extra blank line deleted	0

Conclusions

The method given is a viable technique for ensuring the correct conversion of data onto a new system. The speed of the orginal algorithm was rather slow, about half the speed of a typical Pascal compiler. In consequence, a faster version was written which performs the same algorithm with all 6 bits of a character at once. This technique can be applied to other programming languages, the difficulty being to decide upon what replacements to permit, and how this is to be programmed before application of the check digit algorithm.

Reference

[1] Recommendation V41, `Data Transmission Over Telephone Network' Series V Recommendations, International Telecommunication Union, Geneva, 1977.

[2] Davies, D W, Barber, D L A, Price, W L and Solomonides, C M. Computer Networks and their Protocols. Wiley, 1979.

[3] Thacher, H C. A redundancy check for ALGOL programs. CACM Vol 5, No 6, June 1962.

Complete program listing

```
{ This program produces six 16-bit check digits from a
  a Pascal program. These check digits can be used to
  ensure that the Pascal source text has not been changed.
  The check digits are calculated by using the ISO/CCITT
  cyclic check approved for data transmission and usually
  performed by hardware (some 500 times faster than this
  program).
  For a description of the checking algorithm, see
  D W Davies et al. 'Networks and their Protocols'
  pp263-270.
  Mark 2 version using sets.
}
program checktext(input, output);
const
   linelength = 72;
 { Implementation defined values, to be set for each machine}
   mincharvalue = 0;
   maxcharvalue = 255;
type
   lineindex = 1 .. linelength;
   regindex = 0 .. 16;
   bits = (one, two, three, four, five, six);
   setbits = set of bits;
   ShiftRegister = array [regindex] of setbits;
var
   lpos, lineno: integer;
   firstline, blankline: boolean;
   line: array [lineindex] of char;
   SR: ShiftRegister;
   Convert: array [char] of setbits;

procedure pulse( b: setbits);
   { This algorithm follows Recommendation V41, see
     'Data Transmission Over Telephone Network: Series V
     Recommendations', International Telecommunication Union.
     Geneva (1977).
   }
   var
      i: regindex;
      e: setbits;
   begin
   for i := 15 downto 0 do
      SR[i+1] := SR[i];
   e := SR[16];
```

```
    SR[0] := (b + e) - (b * e);
    SR[5] := (SR[5] + e) - (SR[5] * e);
    SR[12] := (SR[12] + e) - (SR[12] * e);
    end; {pulse}

procedure initialise;
    var
        i: regindex;
        ch: char;
    begin
    for i := 0 to 16 do
        SR[i] := [ ];
    for ch := chr(mincharvalue) to chr(maxcharvalue) do
        Convert[ch] := [ ];
    Convert['a'] := [one];
    Convert['A'] := [one];
    Convert['b'] := [two];
    Convert['B'] := [two];
    Convert['c'] := [one,two];
    Convert['C'] := [one,two];
    Convert['d'] := [three];
    Convert['D'] := [three];
    Convert['e'] := [one,three];
    Convert['E'] := [one,three];
    Convert['f'] := [two,three];
    Convert['F'] := [two,three];
    Convert['g'] := [one,two,three];
    Convert['G'] := [one,two,three];
    Convert['h'] := [four];
    Convert['H'] := [four];
    Convert['i'] := [one,four];
    Convert['I'] := [one,four];
    Convert['j'] := [two,four];
    Convert['J'] := [two,four];
    Convert['k'] := [one,two,four];
    Convert['K'] := [one,two,four];
    Convert['l'] := [three,four];
    Convert['L'] := [three,four];
    Convert['m'] := [one,three,four];
    Convert['M'] := [one,three,four];
    Convert['n'] := [two,three,four];
    Convert['N'] := [two,three,four];
    Convert['o'] := [one,two,three,four];
    Convert['O'] := [one,two,three,four];
    Convert['p'] := [five];
    Convert['P'] := [five];
    Convert['q'] := [one,five];
    Convert['Q'] := [one,five];
    Convert['r'] := [two,five];
    Convert['R'] := [two,five];
    Convert['s'] := [one,two,five];
    Convert['S'] := [one,two,five];
    Convert['t'] := [three,five];
    Convert['T'] := [three,five];
    Convert['u'] := [one,three,five];
```

```
   Convert['U'] := [one,three,five];
   Convert['v'] := [two,three,five];
   Convert['V'] := [two,three,five];
   Convert['w'] := [one,two,three,five];
   Convert['W'] := [one,two,three,five];
   Convert['x'] := [four,five];
   Convert['X'] := [four,five];
   Convert['y'] := [one,four,five];
   Convert['Y'] := [one,four,five];
   Convert['z'] := [two,four,five];
   Convert['Z'] := [two,four,five];
   Convert['0'] := [one,two,four,five];
   Convert['1'] := [three,four,five];
   Convert['2'] := [one,three,four,five];
   Convert['3'] := [two,three,four,five];
   Convert['4'] := [one,two,three,four,five];
   Convert['5'] := [six];
   Convert['6'] := [one,six];
   Convert['7'] := [two,six];
   Convert['8'] := [one,two,six];
   Convert['9'] := [three,six];
   Convert['('] := [one,three,six];
   Convert[')'] := [two,three,six];
   Convert['<'] := [one,two,three,six];
   Convert['>'] := [four,six];
   Convert['+'] := [one,four,six];
   Convert['-'] := [two,four,six];
   Convert['*'] := [one,two,four,six];
   Convert['/'] := [three,four,six];
   Convert['='] := [one,three,four,six];
   Convert['.'] := [two,three,four,six];
   Convert[','] := [one,two,three,four,six];
   Convert[':'] := [five,six];
   Convert[';'] := [one,five,six];
   Convert['↑'] := [two,five,six];
   Convert['@'] := [two,five,six];
   Convert[' '] := [one,two,five,six];
   Convert[''''] := [three,five,six];
   firstline := true;
   lineno := 0;
   end; {initialise}

procedure print;
   procedure printreg( b: bits);
      var
         i: regindex;
      begin
      for i := 0 to 15 do
         if b in SR[i] then
            write( '1' )
         else
            write( '0' );
      write( ' ' );
      end; {printreg}
   begin
```

```
    writeln;
    writeln( 'Check digits are' );
    printreg(one);
    printreg(two);
    printreg(three);
    writeln;
    printreg(four);
    printreg(five);
    printreg(six);
    writeln;
    writeln( 'Total number of lines', lineno);
    end; {print}

procedure readline;
    { Ignores initial spaces,
      Replaces brackets by longer alternatives,
      Checks characters after 'linelength' are spaces,
      Set 'lpos' to ignore trailing spaces,
      Set 'blankline' as necessary.}
    var
       ch: char;
       charsread: integer;
    procedure sub(chx, ch1, ch2: char);
       { Substitute ch1 and ch2 for chx }
       begin
       if ch = chx then
          begin
          if lpos > 72 then
             writeln(' Line too long, line no', lineno)
          else
             begin
             line[lpos-1] := ch1;
             line[lpos] := ch2;
             end;
          lpos := lpos + 1;
          charsread := charsread + 1
          end;
       end;  { sub }

    begin
    lineno := lineno + 1;
    if eoln(input) then
       blankline := true
    else
       begin
       ch := input↑;
       charsread := 0;
       while (ch = ' ') and (not eoln(input)) do
          begin
          get(input);
          charsread := charsread + 1;
          ch := input↑;
          end;
       if eoln(input) then
          blankline := true
```

```
      else
         begin
         blankline := false;
         lpos := 1;
         while not eoln(input) do
            begin
            if lpos <= 72 then
               line[lpos] := ch;
            lpos := lpos + 1;
            { Text to be deleted if curly brackets not supported}
            sub( '{', '(', '*' );
            sub( '}', '*', ')' );
            { End of curly bracket text }
            { Text to be deleted if square brackets not supported}
            sub( '[', '(', '.' );
            sub( ']', '.', ')' );
            { End of square bracket text }
            if (charsread > 72) and (ch <> ' ') then
               writeln( 'Non-spaces after col 72 on line', lineno );
            get(input);
            charsread := charsread + 1;
            ch := input↑;
            end;
         lpos := lpos - 1;
         if lpos > 72 then
            lpos := 72;
         while line[lpos] = ' ' do
            lpos := lpos - 1;
         end;
      end;
   get(input);
   end; {readline}

procedure processline;
   { Checks all characters are valid Pascal characters.
     Call check for each character. }
   var
      i: lineindex;
      j: setbits;
   begin
   for i := 1 to lpos do
      begin
      if firstline then
         write(line[i]);
      j := Convert[line[i]];
      if j = [ ] then
         writeln( 'Non Pascal Character =', line[i],
                  'on line no', lineno)
      else
         pulse(j);
      end;
   if firstline then
      writeln;
   pulse([one,three,five,six]);
   firstline := false;
```

```
   end; {processline}

begin
initialise;
while not eof(input) do
   begin
   readline;
   if not blankline then
      processline;
   end;
print;
end.
```

When run on the ICL 2972 computer without any change to the source text (since ( ) and [ ] are supported and the minimum and maximum character values are as stated), the output was as follows:

```
(* This program produces six 16-bit check digits from a

Check digits are
1100100000001101 1100111010011011 0010110101010101
0001101011001001 1101101110111001 0111111100110011
Total number of lines          278
```

When run on another computer, the unaltered text must be used to confirm the algorithm by execution. Note that the program output may be solely in upper (or lower) case if this permitted conversion is done. The line count may be different if additional blank lines are added. The first and second line of the comment at the start of the program repeats 'a'. This mistake has been left in to trap the unwary who might assume that such small changes would go unnoticed.

This is the second version of the program using sets. On the ICL 2972, the revised version is three times faster than the orginal version. Note that the <> used in the Boolean case must be replaced by set inequality which is (a+b) - (a*b). On the 2972, the compiler takes 2.87 secs to compile this program compared with 1.62 secs to process its own source text. Hence the speed is acceptable for the intended application.

Appendix C
The Assumptions program

The Assumptions program checks that an implementation has the basic facilities necessary for running the tests. These requirements are quite modest. The character set and line length requirements pervade the entire suite but the others are needed only for a minority of tests. For instance, the type real is needed in over 50 tests but in only a few is any assumption made of the accuracy of at least four digits. In practice, the most troublesome one with existing compilers is likely to be that of local files. Over 50 tests depend upon this although it would be simple to change these to program parameters (except, of course, for formal validation where no changes can be made). The portability of programs involving sets is a well-known problem so that the test suite uses sets very sparingly.

The program can be typed in by hand before obtaining the whole suite to locate potential difficulties.

```
{ The purpose of this program is to check that the
  basic requirements for running the test suite are met by
  an implementation. It does not form part of the test
  suite, but is an aid to avoid problems with the suite
  when running all the programs.
 }
program Assumptions(output);

const
   curlybracket = '{';
   squarebracket = '[';
   basicset = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789( )<>+-*/=.,:;↑ ''';

type
   charset = packed array [ 1 .. 52 ] of char;
   reason = packed array [ 1 .. 20 ] of char;
   smallset = set of 0 .. 15;

var
   chars: charset;
   i, j: 1 .. 52;
   sset: smallset;
   localfile: text;
   max: integer;
   one: real;

   procedure fail( s: reason);
      begin
      writeln;
      writeln( 'BASIC REQUIREMENT NOT MET: REASON');
      writeln( '      ', s);
      end;

begin
```

```
{ check all basic characters distinct }
chars := basicset;
for i := 1 to 51 do
   for j := i + 1 to 52 do
      if chars[i] = chars[j] then
         fail( 'CHAR SET TOO SMALL  ' );

{ check line length of 72 characters accepted }
i :=
000000000000000000000000000000000000000000000000000000000000000000000000000001
;
if i <> 1 then
   fail( 'LINE LENGTH < 72    ' );

{ check line length of 72 accepted, constructed by substitution }
{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{{1}
i := 2; { in case closing bracket missed }
if i <> 2 then
   fail( 'LINE   (* < 72 CHARS' );

{ check 72 characters accepted as output to textfile }
writeln( 'CHECK NEXT LINE HAS 72 CHARS' );
for i := 1 to 9 do
   write( '12345678' );
writeln;
writeln( 'END OF LINE' );

{ check maxint big enough }
if maxint <= 32000 then
   fail( 'MAXINT IS TOO SMALL ' );

{ check reals are accurate enough }
if 1.001 <= 1.0 then
   fail( 'REAL TOO INACCURATE ' );

{ Check that small sets are permitted }
sset := [ 0 .. 15 ];
sset := sset - [ 1 .. 14 ];
if sset <> [ 0 , 15 ] then
   fail( 'SMALL SETS WRONG    ' );

{ Check local files implemented }
rewrite(localfile);
writeln(localfile, maxint, '  ', 1.0);
reset(localfile);
read(localfile, max, one);
if (max <> maxint) or (one <> 1.0) then
   fail( 'LOCAL FILES WRONG   ' );

writeln( ' CURLY BRACKET IS ', curlybracket);
writeln( 'SQUARE BRACKET IS ', squarebracket);
end.
```

Index

Accreditation 90, 93, 138
Accuracy 2, 81, 90, 100, 108
ACVO 105
Ada 32, 44, 80, 82, 88, 96, 98, 101, 103, 104, 105, 123
Ada Compiler Validation Capability 104
Addyman, A.M. 1, 4, 6, 13, 14, 24, 139
ad-hoc tests 40
ADPE Selection Office 119
AFSC 15, 25
Agence de l'Informatique 75
Air Navigation Order 138
Airworthiness 35
Airworthiness certificate 93, 135
Airworthiness Requirements Board 136
Algol 48, 59, 100, 120
Algol-60 4, 15, 96, 105
Algol-68 12, 106
'Almost working' programs 131, 133
Alterations 159
Alternative strategies 101
Alternative symbols 66
Ambiguities 4, 96, 112
Analysis of results 81
Anomalous cases 99
ANSI 82, 88, 95, 106-9, 111, 112
Appeal procedure 68
Architecture 82, 101
Arithmetic operations 124
Array 107
ASCII 113
Asre'ment system 84
Assessment 70, 71, 84
Assignment compatibility 11
Assignment statement 21
Assumptions 26, 64, 169
Attention List 6
Attribute grammars 121
Authentication 73
Automatic analysis 83

Babel 125, 133
Back-erd 121, 122
Backus 17
Bailey, C.B. 98, 100, 139
Baird, G.N. 108, 139, 140
Ballot 83
Barnes, J.G.P. 121, 139, 142
BASIC 35, 106, 107, 122
Basic Pearl 121
BCPL 101
Belfast compiler 89
Bell Laboratories 32
Belz 103
Benchmark tests 109
Benefits 73
Bjorner, D. 103, 140
Black box 91, 94, 127
Blake, F.M. 139
BNI 75
Boolean-type 20
Bowden, J.S. 140
Brauer, W. 140
British Civil Airworthiness Requirements 135
Brooks, N.B. 141
Brown, P.R. 108, 139
Brown, W.S. 140
BS 6192 8, 11, 14, 25
BSI 4, 68, 69, 84, 86-88
BSI Test House 71
Byrne, B.A. 59

Campbell, I. 80
Canada 69
CCITT 159
CDC 6600/7600 98
CELAR 75, 121
CERCI 75
Certification 68, 69, 79, 84, 86, 87, 92, 93, 109

Challenge 46
Character-constant 16
Charter, J.W. 139
Char-type 19
Check digit algorithm 162
Check sum 160
Chemla, P. 80
CHILL 32, 103
Cichelli, R.J. 139
Ciechanowicz, Z.J. 90
Civil Aviation Authority 93, 135-38
Claims 28
CNAM 75
CNET 75
COBOL 1, 5, 8, 15, 27, 35, 38, 68, 71, 82, 88, 90, 95, 103, 105, 108-11, 114, 122
Cody, W.J. 32, 34, 97, 140,150
Communications facility 110
Compatibility 82
Competence 138
Compile-time 48
Compile-time switch 95
Complex systems 91
Complexity 2
Complexity tests 78
Compliance 90
Conditional tests 40
Conformance 18
Conformance tests 37, 84, 85, 109, 145, 146
Conformant arrays 7, 9, 10, 31, 32, 47, 82, 83, 155, 157
  innermost dimension packed 10
Conformity 72
Constructor 20
Contractual certainty 59
Control Data Algol 68 compiler 106
Conversion 26, 64
Coral 123
Coral-66 71, 85, 90, 103, 115-17
'Correct' program 133
CSA 69
Cugini, J.V. 108, 139, 140
Curnow, H.J. 34

Dangling pointers 56
DAVE 133
Davis, A.M. 141
de facto definition 94
Default file 16
Deficiencies 120
Dekker, T.J. 98, 141
Demise 43
DEMKO 69
DeMorgan, R.M. 5, 25, 141
Detection 112
Deviance 18
Deviance tests 38, 85, 147, 148
Diagnostic package 46
Dialects 44, 45
Directives 78
DIS/5 6
Disagreements 113
Div test 62
DoD 88, 104, 109, 120
Double-length library functions 98
DPS/13/4 6
DRS 61
du Croz, J. 34
Dyadic 20

ECE 70
ECMA TC21 106
EEC 70, 85, 86
Efficiency 99, 115, 116, 119, 120
Engineering 75
England, C. 24
Enhancements 117
Environment effect 116
Error handling tests 149
Errors 2, 4, 12, 19, 21, 41, 53, 55, 57, 81, 94, 95, 112, 116, 120, 127
Euclid 103
Europe 80, 87
EUROSOFT 75
Evaluation 115, 116
Existence errors 57
ExpDigits 20
Experts Group Meeting 7
Extension to Pascal 9
Extension tests 158
Extensions 19, 22, 59, 82, 92, 95, 96, 117, 118

Failure tolerance 126
Faithfulness requirements 98
FAR 136
Faults 138

FCTC 67, 68, 71, 87, 105, 108-11, 113, 114, 119
  charges 68
  description of service 108
FCVS 118, 119, 127, 128
Feldman, S.I. 140
File access 78
File-type 20
Files 107
Findlay, W. 58
Fischer, C.N. 13, 14, 58
Fixed formats 124
Flag ability 95
Floating-point arithmetic 92, 98, 123
Floating-point data types 111
Floating-point formats 124
Folk-lore 59
For-loops 22, 31, 62
For-statement 8, 50
Formal 102
Formal verification 33
Forrester, M.H. 139
FORTRAN 1, 5, 8, 10, 27, 32, 35, 38, 48, 59, 68, 71, 82, 90, 101, 103, 105, 107, 114, 117, 122, 123, 133
Fortran-77 96, 97, 119
Fosdick, L.D. 133, 143
Foster, K.A. 103, 141
'Fragile' programs 131, 133
France 75
Freak, R.A. 24, 143
French Ministry of Defense 121
Front-end 121
Function-designator 20
Future work 13

Gannons, C. 141
Garwick, J.V. 101, 141
Gerhart, S.L. 103, 141
Gien, M. 75, 80
Gilsinn, D.E. 106, 141
Glasgow University 46
Goodenough, J.B. 34, 103, 139, 141, 144
Goos, G. 139
Goto-statement 8
Grading 84
Gresham's Law 59
Grune, D. 139, 142
GSA 5, 88, 109
Guidelines 8
Guinea-pig 73
Gutfeldt, H. 34

Hardware errors 127
Hay, A. 46
Header comment 17
Heath, A.J. 139
Heliard, J.C. 142
High-level languages 90
Hill, I.D. 5, 13, 25, 34, 141
Hissen, A. 14
Holberton, F.E. 117, 142
Honeywell/Level-6 75
Hopper, G.M. 109
Hoyt, P.M. 117, 119, 142
Hruschka, P. 121
Huenke, H. 139

ICAO 136
Ichbiah, J. 142
ICL 60, 89
Identified-variables 13
IECCA 115-17
IEEE 31, 142
IFIP Working Group 2.1 106
'Illegal' programs 130, 133
Implementation defined 19, 78
Implementation-defined tests 153-55
Implementation dependent 20, 41
Implementers 86
Implementers' guide 104
Incidents 138
Index-expressions 20
Infinity 22
Input/output 59, 124
  interactive 12
INRIA 75
Inspection 139
Integer-type 19
Integrity 64
Intel/8086 75
Interaction 92
  between language elements 110
Interactive I/O 12
Intermediate language 122
Interpretation 69, 85
Irregular tests 157
ISO 4, 83, 87, 106, 134, 260
ISPRA 88

Japan 8
JAR-25 135
Jensen, K. 25, 33, 58
Job-control language 99
Joint Airworthiness Requirements 136
Jones, B. 5, 25, 144
Jones, C. 142
Jones, R.E. 100, 139
Jovial 120

KDF9 125
Kelly, J.R. 144
Kemeny, Prof. 106
Kernel 79
Key value 56
Kirkham, C.C. 6, 14
Kitemark 69
Krieg-Brueckner, B. 142
Kronental, M. 80
Kurtz, Prof. 106

Label/goto 51
Lazy I/O 12
LeBlanc, R.J. 13, 14, 58
Legal action 81
Level 0 10
Level 1 10, 22, 155
Licences 70
Limitations 74
Line length 30
Local files 81, 169
Logarithmic representation 31
Lomuto, N. 144
LTR 75, 121

Machine dependent features 120
Machine dependent I/O 116
Magnetic tape testing 87
Main-frame 1, 82, 89
Malagardis, N. 80, 139
Malcolm, M.A. 142
Malcolm's algorithm 98
Management 42
Mandl, R. 144
Manufacturer's view 59
Market needs 72
Mathematisch Centrum 106
Maurice, P. 80
Maxint 19
Meek, B.L. 13
Member-designator 20
Mickel, A.B. 58
Microprocessors 1, 83
Miner, J.F. 12, 14, 24
Mixed-case 16
MoD 71, 85, 90, 93, 103, 115, 116, 138
Mod test 62
Model implementation 46
Modification 87
Monitoring 126
Morgan, L. 64, 84
Motorola/68000 75
Multi-access 99
Multi-dimensional arrays 9
Multi-level 47
Multics Pascal 32

N462 7, 9
N510 7, 9
N595 11
N678 8
National Computing Centre 5, 108
National Semiconductor/16000 75
National Technical Information Service 108
Navy Programming Languages Group 109
NBS 106, 107, 117
Near misses 127
NEMKO 69
'Nonsense' programs 130, 133
Non-standard circumstances 107
Non-standard compilers 81, 96
Non-standard facility 118

Obscure issues 30
Octal value 124
Ollivier, G. 80
Optimization 100, 122
Optimizing compilers 92, 105
Optional facilities 92
Osterweil, L.J. 133, 143
Ostrand, T.J. 103, 144
Overflow 122
Overlaying 116

Pack procedure 21
Packed arrays 9
Page procedure 20
Parity check 159

Parker, E.G. 117, 142
Pascal News 35, 85
Pascal Test Suite 145
P-code 83
Pearl 121. 122
Performance 72
PERQ 60
Peterson, J.L. 143
Petri, C.A. 143
Petri nets 102, 121
Planning 64
PL/I 10
Pointer errors 55
Polak, W. 33, 34
Portability 1, 73, 75, 90, 95, 102, 169
Preliminary Ada Reference Manual 134
Problems 72
Procedure-statement 21
Process control 126
Procurement 109
Production-standard compiler 111
Program parameters 21
Proof 35, 103, 127
Pseudo-random tests 123
Pyle, Prof. 127

Quality 2, 21, 31, 92, 98
Quality assurance 69, 139
Quality Assurance Council 71
Quality control 69, 93
Quality systems 71
Quality tests 41, 150-52
Queen's University of Belfast 46
Quinn, C. 58

Randle, Prof. 127
Random tests 100
Rauscher, T.G. 141
Read-only 10
Real-time processing 107
Real-type 20, 169
Reassessment 93
Recovery 3
Recursion 116
Recursive production 22
Redundancy 126
Registration 70
Reimheer, G.W. 143
Reinsch, C.H. 98, 143
Reports 22, 76, 84-86, 88, 105, 119
Reserved words 30
Retesting 91, 95, 117
Revalidation 68, 110, 111
Robinson, R.A. 143
Robustness 99
Roubine, O. 142, 143
Rounding algorithm 115
RSRE 71, 115-17
RTL/2 101, 121, 122
Rules 73
Run-time 53, 122

Safety Mark 70
Sale, A.H.J. 4, 14, 15, 24, 25, 35, 50, 58, 80, 89, 139, 143, 144
Samples 145
San Diego draft 7
Sare, J.B. 14
Satellite validation facility 88
Saville, N. 24
Scandinavian market 69
Schryer, N.L. 34
Scope rules 11
Scowen, R.S. 90, 144
Self-checking 112
SEMA 75
Semanol 103
Semantics 107
SEMKO 69
SEMSID 117
Sems/Mitra 75
Service operation 64, 84
Sets 169
Sheppard, C.L. 106, 141
Shift resisters 160
Side-effects 40
Sidi, J. 75
SIGNUM Newsletter 98
Simms, K.A. 33
Simulation 98
Single-length library functions 98
Size requirements 120
Skall, M.W. 140
SoftTech 104
Software management 139
SOL 75
SOL Club 76
Speed of compilation 112
Speed of element access 48
Standard functions 123, 124
Standardized report 67

Static checker 46
STERIA 75
Stevenson, D. 34
Storage overhead 49
Strachey, C. 103, 144
Strait, J.P. 58
String-character 19
String-constant 16
String manipulation 107
Strings 9
Structural type equivalence 156
Structure 16
Stuggart University 121
Subroutines 96
Substitutions 26
Support 88
'Suspicious' programs 133
Syntax errors 120
Syntax rules 11, 15
SYSECA 75

Tailoring a program 109, 113
Tennent, R.D. 24
Test problems 30
Test-tester 117
Testing 70, 90, 91
Testing centre 68
Testing procedures 26
Three Rivers Corporation 60
Time overhead 49
Tools 75
Total Width 19, 20
Translator box 45
Transmission of digital data on telephone lines 159
TTY compatibility 88
Tuning 100
Type Certificate 136
Type rules 11

Unassigned variable 134
Undefined values 54, 98, 125
Uniformity 73
United Kingdom 80, 85, 88, 137, 138
United States 71, 82, 87, 88, 90, 95, 103
Unix-like system 75
Unpack procedure 21
Unpacked arrays 9
Untestability 72
Urban, R.J. 141
USAF 108, 109
User-friendly 64
User guide 64
US Navy 117, 118

Vagueness of Standards 96
Validation 92
Validation suite 15, 22, 35, 81
Variant errors 55
Version number 16
Violation 12
  of explicit prohibitions of standard 114

Waite, W.M. 32, 34, 97, 140, 150
Warning capability 96
Warning messages 130
Warshall, S. 144
Watt, D.A. 58
Wells, T.D. 139
Welsh, J. 34, 46, 58
Weyuker, E.J. 103, 144
White, T.A.D. 115
Wichmann, B.A. 1, 4, 5, 15, 24, 25, 33, 34, 80, 139, 141, 142, 144, 159
Williams, A.M. 63
Williams, D.R. 143
Wilson, I.R. 14
Wirth, N. 4, 5, 9, 25, 33, 58
Width-statement 8
Word-length 59
World-wide service 87
'Wrong' program 132, 133

X3J2 106, 107
X3J4 108, 112
X3J9 7

ISBN 0 471 90133 4

WICHMANN
CIECHANOWICZ
(EDITORS)

PASCAL COMPILER VALIDATION