A Pascal processor validation suite

B. A. WICHMANN* and A. H. J. SALE†

March 1980, NPL Report CSU 7

Abstract

The report is the documentation accompanying a series of test programs written in Pascal. The suite of programs may be used to validate a Pascal processor by presenting it with a series of programs which it should, or should not, accept. The suite also contains a number of programs that explore implementation-defined features and the quality of the processor. The tests are generally based on the draft ISO Standard for Pascal.

This is a working document. It is being continually revised and extended. Comments, corrections, extra tests, and results of running any tests would be most welcome.

This is version 2.3, an earlier edition was issued as a report from the University of Tasmania.

1 INTRODUCTION AND PURPOSE

This paper describes a suite of test programs which has been designed to support the draft Standard [1, 3] for the programming language Pascal (Jensen & Wirth, 1975) prepared for approval by ISO[1]. It therefore follows similar work done by AFSC for COBOL[2], and Wichmann for Algol 60 [7, 5, 8].

The suite of programs is called a validation suite for Pascal processors; however it is important to emphasize that no amount of testing can assure 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 draft 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.

*Computing Services Unit, NPL

†Department of Information Science University of Tasmania, GPO Box 252C, HOBART, Tasmania 7001

[1]In the rest of this paper, the draft Standard is simply referred to as the Standard.

On the other hand, the test series 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 large number of tests, and their independent origin, will assist in detecting many probable implementation errors. The series 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. Most of the following rules apply to all programs: a few hold everywhere except in a few test cases meant to test the particular feature involved. Such rules are marked by an asterisk and a following note points out the exceptions.

- Each program starts with a header comment, whose structure is given later.
- The header comment is always immediately followed by an explanatory comment in plain English, which describes the test to be carried out and its probable results.
- 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.
- All program lines are limited to 72 character positions.
- * 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- 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.)

- 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.
- * 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 and others which serve purely as a cross reference).

2.1 The header comment

The header comment always begins with the characters "TEST" in positions 1-5 of a line. No other comments are permitted to have the character "{" and "T" directly juxtaposed in this way. The syntax of a header comment in Extended Backus Naur Form [4] is given by:

```
header-comment = "{TEST" program-number "," "CLASS=" category-name "}" .

program-number = number {"." number} "-" number .

number = digit {digit} .

category-name = "CONFORMANCE" | "DEVIANCE" | "IMPLEMENTATIONDEFINED" |
                "ERRORHANDLING" | "QUALITY" | "EXTENSION" .
```

For example, a possible header comment is:

```
{TEST 6.5.3-10, CLASS=CONFORMANCE}
```

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 title is constructed from the section number by replacing "TEST" by "t", "." by "p" for point, and "-" by "d" for dash. Thus the above header comment belongs to a program t6p5p3d10. 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 depends 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" 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.

2.2 The program classes

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 test 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:

- the processor handles an extension of Pascal,
- the processor fails to check or limit some Pascal feature appropriately, or
- 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, and to generate a short series where one conformance test collects several examples.

2.2.3 Class = IMPLEMENTATIONDEFINED

In some sections of the Standard, implementors are permitted to exercise some freedom in implementing a feature. An example is the significance limit of identifiers; another is the evaluation order of boolean expressions. The CLASS = IMPLEMENTATIONDEFINED tests 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 implementation dependency. The collection of such implementation dependencies is useful

to the writers of portable software. Some tests in this category require care in interpretation, as the messages generated by the test program rely on some assumptions about the processor implementation. The programs may or may not be standard Pascal: often they are not.

For example, one program attempts to measure the significance limits of identifiers by declaring a series of differing length in an inner procedure that are different from an outer series by their last letter. Thus it violates the requirement for uniqueness over the first eight characters and relies on masquerading redefinition under the scope rules for its effect. One processor, however, reports that just this is happening during compilation. Though this is ideal behaviour, it would destroy the test if the program then was not permitted to run. (In this case, in fact, the messages were only warnings.)

2.2.4 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.

2.2.5 Class = QUALITY

These tests are a miscellany of test 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 have known syntax errors which can be used to inspect the diagnostics.
- 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.

2.2.6 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. In this case, the class in the header comment is followed by a sub-class, as in the example:

```
{TEST 6.8.1-1, CLASS=EXTENSION, SUBCLASS=CONFORMANCE}
```

The subclass gives the purpose of the test according to the previously explained classes.

2.2.7 Class = DUBIOUS

The definition of a standard is envitably incomplete. Sometimes, such as with the definition of floating point, such incompleteness is deliberate while in other cases it is clearly accidental. Programs whose interpretation is in doubt are therefore added to this class. One hpoes that the current revision of the draft BSI Standard by ISO will resolve the ambiguities in many of these programs.

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 previous file.
3. A copy of this report.
4. A file consisting of the sequence of test programs arranged in lexicographic order of their program-number (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). It is documented in an Appendix.

4 REPORTING THE RESULTS

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

```
PASCAL PROCESSOR IDENTIFICATION
      MACHINE:
      COMPILER:
------------------------------
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
--------------
Number of errors correctly detected = ?
Number of errors not detected = ?

Details of errors not detected:
       .....

IMPLEMENTATION-DEPENDENCE
-------------------------
Number of tests run = ?
Number of tests incorrectly handled = ?
      Details of implementation-dependence:
         .....

QUALITY MEASUREMENT
-------------------
Number of tests run = ?
Number of tests incorrectly handled = ?
       Results of tests:
```

```
        .....

EXTENSIONS
----------
Extensions present = ?
```

4.1 PDP11 results

```
PASCAL PROCESSOR IDENTIFICATION
MACHINE: DEC PDP-11 running RSTS V06C-03
COMPILER: OMSI Pascal-1 (Field Test Version X1.2)
--------------
TEST CONDITIONS
     DATE: 9 and 10th September 1979.
     TESTS BY: Barry Smith, Oregon Software
               (Implementation/maintenance team)
     TEST VERSION: 2.1
     REPORTED BY: A H J Sale (13th September 1979)
---------------
CONFORMANCE TESTS
----------------
Number of tests attempted = 137
Number of tests passed    = 122
Number of tests failed    =  15 (13 causes)
Invalid tests discovered  =   2

6.1.8-3 Comment delimiters are not required to be pairwise
matching; this makes { This part of the scanner looks
for a *) delimiter} a disallowed comment.

6.2.2-3 Pointer scope is not handled correctly, so that
programs fail to compile.

6.2.2-8 Assignment to function-identifier from within
nested function generates bad code.

6.4.3.3-1 Empty record types with semicolons and empty case
variants are not permitted.

6.4.3.5-2 and -3. An unknown interaction between RSTS I/O
on temporary files and the implementation of the
run-time support.

6.4.5-9 Equal compatible sets of different basetypes do not
compare equal. (Pascal-1 scales the basetype to force a
representation of bit 0 on the lowerbound, giving errors
in comparisons as shifts are not inserted to
compensate. Also set of char is implemented as set of
' ' .. '~' (64 chars).)

6.6.3.1-5 and 6.6.3.4-2. Only Jensen and Wirth procedural
parameters allowed, not the BSI version. The second test
is relevant to the feature actually implemented, but has
not been run with modifications.
```

6.6.5.2-3 Does not check eof on an empty temporary file.

6.6.5.4-1 Pack and unpack not implemented.

6.8.2.1-1 Empty field specifications not allowed in record declarations.

6.9-1 Eoln and eof not correct: relation between RSTS and implementation causes unknown fault.

6.9.2-3 Conversions on reading real numbers are not identical to the conversions performed by the compiler.

6.9.4-7 Writing boolean values is incorrectly right-justified. (Sale comments that the new draft may change this or tighten up wording.)

DEVIANCE TESTS

Number of test attempted = 95
Number of tests showing true extensions = 43 (25 cases)
Number of tests not detecting erroneous deviations = 52
Number of claimed extensions = 2 (as stated by B Smith)

6.1.5-4 Allows real number constants without digits after point.

6.1.7-5 and 6.9.4-12. Packed is ignored so that packed array of char is identical to array of char, and similarly with other structures.

6.1.7-6 and -7. The requirements to be a string-type are not checked, allowing deviant programs to execute.

6.1. 7-8 The requirements to be a string-type are not checked, together with an obvious error, allows erroneous values to be given to a type.

6.1.7-11 Allows empty string: ie '' is equivalent to packed array[1..0] of char.

6.2.2-4 Incorrect scope allows incorrect program to compile.

6.2.2-7 Invalid program executes with (a) function whose identifier is inaccessible and (b) another function has an attempted assignment outside its block.

6.2.2-9 A function-identifier may be assigned to outside its block.

6.3-2, -3, -4, -5, 6.7.2.2-9. Signed characters, strings and enumerated types are allowed.

6.4.3.1-1 Allows packed scalars, subranges, ie not
restricted to structures.

6.4.3.1-2 Allows "packed" type-identifier.

6.4.3.2-5 String types are allowed to have non-integer
subrange indextypes.

6.4.3.4-3 Set of real erroneously not detected.

6.4.5-2 Var parameters which are compatible but not
identical are allowed.

6.4.5-3 and -13. Non-identical array types allowed as var
parameters.

6.4.5-5 Non-identical pointer types allowed as var
parameters.

6.4.6-10, -11 and -12. Compiles file assignment as
descriptor copy, and similarly for records containing
file components.

6.6.2-5 Allows function definitions without any assignment
to function-identifier.

6.8.2.4-2, -3 and -4. Allows goto statement to transfer
into structured statement components.

6.8.3.9-2, -3, -4 and -16. Any assignment to a
for-control-variable is allowed inside the controlled
statement, and it in fact changes the value.

6.8.3.9-19. Two loops using same variable interact to
produce infinite loop construction, or other
insecurities.

6.8.3.9-9, -13 and -14. Allows a for-control-variable to be
program-global, non-local, and other insecurities.

6.10-1 Ignore program parameters, allowing use of external
file not stated.

6.10-3 The files input and output are not implicitly
declared at the program level, but at a lexically
enclosing level.

6.10-4 The entire program heading, including the reserved
word program, may be omitted.

ERROR-HANDLING

Number of tests attempted = 48

Number of errors correctly detected = 11 (9 causes)
Number of errors not detected = 2

Tests failed

6.4.3.3-12 Crash at run-time due to empty record-field.

6.6.5.2-2 Relation between RSTS I/O and implementation
run-time support.

Tests passed and errors detected

6.4.6-6, 6.6.3.2-1 Assignment compatibility: indextype vs
subscript value.

6.6.5.2-1 Put not allowed if eof false.

6.6.6.2-4 ln(0.0) or ln(negative).

6.6.6.2-5 sqrt(negative), but continues execution.

6.6.6.2-2 trunc(largereal)

6.6.6.2-3 round(largereal)

6.7.2.2-3, -8 Div and mod by 0, but continues execution.

6.9.2-4 Read of textfile, but chars do not represent
integer value.

6.~.2-5 Read of textfile, but chars do not represent real
value.

Errors not detected

Use of undefined value.

Variant undefinition.

All assignment compatibility except indextype in arrays.

Nil or undefined pointer deferencing.

Undefined function result.

Fill buffer aliasing and use of file.

Dispose of nil or undefined pointer value.

Dispose of variable currently var parameter or with
aliased.

Dynamic variant record used in expression or assignment.

Succ or pred of limiting value in type.

Chr of very large integer.

Overflow of integer type.

Assignment compatibility with overlapping sets.

Case expression with no matching label (falls through).

Use of for-control-variable after loop termination.

nested loops using same control-variable.

IMPLEMENTATION-DEPENDENCE

Number of tests run = ?

Number of tests incorrectly handled = ?

Maxint = 32767.

Set of char not implemented, but taken as equivalent to set
of ' ' .. ' '.
set limits are 0 .. 63.

Standard functions not allowed as functional parameters.

Real representation has 24-bit mantissa, rounds on
arithmetic, eps=5.96e-8; xmin = 2.9e-39; xmax = 1.70e38.

Full evaluation of boolean expressions.

Selection then evaluation in a[i] := exp.

Evaluation then referencing in p^ := exp.

Writes two exponent digits in real numbers.

Default widths: integer 7, boolean 5, real 13.

Rewrite permitted on output.

Both comment delimiters allowed, no others.

QUALITY TESTS

Number of tests run = 24
Number of tests incorrectly handled = 3
Tests failed

6.2.1-9 Compiler loops when presented with program with 50
labels.

6.6.6.2-9 Compiler refuses to compile a real expression in sin/cos test due to "lack of registers".

6.8.3.9-20 Compiler crashes after compiling 11 nested for-loops.

Quality measurements

5.2.2-1 and 6.1.3-3. Any length identifiers allowed: disallow all mis-spellings.

6.1.1-4 Allowable number of types >=50.

6.4.3.2-4 Array[integer] diagnosed but message not good.

6.4.3.3-9 Record fields allocated representation space in declared order.

6.4.3.4-5 Warshall's algorithm timing/space test not yet run.

6.5.1-2 Allowable number of variable declarations >=100.

6.6.1-7 Allowable number of nested procedures must be <=10.

6.6.6.2-6, -7, -8 and -9. Quality tests on sqrt, arctan, exp and ln carried out. Some minor inconsistencies.

6.7.2.2-4 Mod inconsistently implemented for negative operands.

6.8.3.5-2 No warnings for impossible case clauses.

6.8.3.5-8 Allowable number of case-constants >=256.

6.5.3.9-18 Undefined (out-of-range) values of case expressions are possible and are undetected but do no violent damage.

6.8.3.10-7 Allowable number of nested with-statements must be <=3.

6.9. 4-11 Recursive I/O allowed on some file, and still works.

EXTENSIONS

Claimed extensions

6.7.2.3-4 And, or and not are overloaded to be representation-dependent set of operators on integer type.

```
6.9.4-9 Negative field widths in writes of integer produces
octal interpretation in field of abs(width).
```

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 [7] and A.H.J. Sale [6] and significant contributions have also been made by A.M. Addyman and R.D. Tennent. N. Saville 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 draft ISO Standard, nor without the support of the Pascal Users Group.

References

[1] Addyman, A.M. (1979): BSI/ISO Working Draft of Standard Pascal by the BSI DPS/13/4 Working Group, Pascal News No 14, January 1979, pp4-54 and Software - Practice and Experience, Vol 9 pp381-424.

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

[3] British Standards Institute(BSI, 1979a), Draft Standard Specification for the computer programming language Pascal. Document 79/60528 DC, March 1979, also issued as a draft ISO Standard. (The technical content of this is the same as Addyman 1979, but the section numbers differ. Version 2.1 of the kit used Addyman, but 2.2 uses the standard numbering.)

[4] British Standards Institute(BSI, 1979b), Draft Standard Syntactic Metalanguage. Document 79/64467 DC, November 1979.

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

[6] Sale, A.H.J. (1978): Pascal Compatibility Report (revision 2), Department of Information Science Report r78-3, University of Tasmania, May 1978. obsoleted by this document

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

[8] Wichmann, B.A. & Jones, B. (1976): Testing ALGOL 60 Compilers, Software - Practice and Experience, Vol 6 pp261-270. 1976.

A Document details, Not published

Appendix to CSU7 not included in this transciption. Scanned and converted to LaTeX, January 2000.

A.1 Historical note

The general strategy worked well, with strong support by BSI who launched a commercial service. However, neither the UK Government nor Europe provided any encouragement for the service. In consequence, the service depended entirely on the requirements for validation by the US for Federal procurement.

Technically, some changes were made to the suite, but these were mainly cosmetic. NIST objected to the quality tests — the solution being to retain them, but not require that they be passed for successful validation. Commercially, since the validation reports were published, the companies ensured they were passed.

The Pascal standard suffered from lack of support in the US due to the controversy about conformant arrays (optional in the standard). Without this facility, the language does not have the functionality of Fortran and hence lacked credibility.

The design of the PVS was vindicated by the Ada suite, which copied it. The Ada suite was heavily backed by the US Government and the Defence industry and in consequence, was more successful. (NPL contributed a small part of the Ada suite under a contract.) The Modula-2 validation suite was a straight copy of the PVS, but died with the lack of interest in the language.

The Plum-Hall C validation suite has a slightly different design, reflecting that the language is designed round subroutines, rather than programs. NPL added tests for the new mathematical functions which were standardized recently.