Pascal News

October, 1979  Number 16

SPECIAL ISSUE ON THE PASCAL VALIDATION SUITE
COMMUNICATIONS ABOUT THE PROGRAMMING LANGUAGE PASCAL BY PASCALERS

Front Cover  The Twin Cities of Minneapolis and St. Paul say Goodbye to Pascal News

0  POLICY: Pascal News
1  ALL-PURPOSE COUPON
3  EDITOR'S CONTRIBUTION
5  INTRODUCTION TO THE SPECIAL ISSUE
   The Pascal Validation Suite - Aims and Methods by Arthur Sale
10  THE PASCAL VALIDATION SUITE - VERSION 2.2
10  Distribution Information
11  Distribution Format and Addresses
12  A Pascal Processor Validation Suite by B.A. Wichmann and A.H.J. Sale
25  Listing of the Test Programs
142  FOUR SAMPLE VALIDATION REPORTS
142  Introduction
143  Report on University of California compiler for B6700
146  Report on University of Tasmania compiler for B6700
148  Report on OMSI Pascal-1 for PDP-11
151  Report on Pascal-P4
154  Stamp Out Bugs
155  POLICY: Pascal User's Group

Back Cover  Golden Ratio
POLICY: Pascal News (79/09/01)

* Pascal News is the official but informal publication of the User's Group.

Pascal News contains all we (the editors) know about Pascal; we use it as the vehicle to answer all inquiries because our physical energy and resources for answering individual requests are finite. As PUG grows, we unfortunately succumb to the reality of (1) having to insist that people who need to know "about Pascal" join PUG and read Pascal News - that is why we spend time to produce it! and (2) refusing to return phone calls or answer letters full of questions - we will pass the questions on to the readership of Pascal News. Please understand what the collective effect of individual inquiries has at the "concentrators" (our phones and mailboxes). We are trying honestly to say: "we cannot promise more than we can do."

* An attempt is made to produce Pascal News 3 or 4 times during an academic year from July 1 to June 30; usually September, November, February, and May.

* ALL THE NEWS THAT FITS, WE PRINT. Please send material (brevity is a virtue) for Pascal News single-spaced and camera-ready (use dark ribbon and 18.5 cm lines!).

* Remember: ALL LETTERS TO US WILL BE PRINTED UNLESS THEY CONTAIN A REQUEST TO THE CONTRARY.

* Pascal News is divided into flexible sections:

**POLICY** - tries to explain the way we do things (ALL-PURPOSE COUPON, etc.).

**EDITOR'S CONTRIBUTION** - passes along the opinion and point of view of the editor together with changes in the mechanics of PUG operation, etc.

**HERE AND THERE WITH PASCAL** - presents news from people, conference announcements and reports, new books and articles (including reviews), notices of Pascal in the news, history, membership rosters, etc.

**APPLICATIONS** - presents and documents source programs written in Pascal for various algorithms, and software tools for a Pascal environment; news of significant applications programs. Also critiques regarding program/algorithm certification, performance, standards conformance, style, output convenience, and general design.

**ARTICLES** - contains formal, submitted contributions (such as Pascal philosophy, use of Pascal as a teaching tool, use of Pascal at different computer installations, how to promote Pascal, etc.)

**OPEN FORUM FOR MEMBERS** - contains short, informal correspondence among members which is of interest to the readership of Pascal News.

**IMPLEMENTATION NOTES** - reports news of Pascal implementations: contacts for maintainers, implementors, distributors, and documentors of various implementations as well as where to send bug reports. Qualitative and quantitative descriptions and comparisons of various implementations are publicized. Sections contain information about Portable Pascals, Pascal Variants, Feature-Implementation Notes, and Machine-Dependent Implementations.

* Volunteer editors are (addresses in the respective sections of Pascal News):

Andy Mickel - Outgoing editor; Rick Shaw - Incoming editor
John Eisenberg - Here and There editor
Rich Stevens - Books and Articles editor
Bob Dietrich and Gregg Marshall - Implementation Notes editors
Jim Miner and Tony Addyman - Standards editors
Andy Mickel and Rich Cichelli - Applications editors
Jenny Sinclair and Rick Marcus - Tasks editors
Pascal User's Group, c/o Rick Shaw  
Digital Equipment Corporation  
5775 Peachtree Dunwoody Road  
Atlanta, Georgia 30342 USA

**NOTE**

- Membership is for an academic year (ending June 30th).
- Membership fee and All Purpose Coupon is sent to your Regional Representative.
- See the Policy section on the reverse side for prices and alternate address if you are located in the European or Australasian Regions.
- Membership and Renewal are the same price.
- The U. S. Postal Service does not forward Pascal News.

[ ] Enter me as a new member for:  [ ] 1 year ending June 30, 1980
[ ] Renew my subscription for:  [ ] 2 years ending June 30, 1981
[ ] 3 years ending June 30, 1982

[ ] Send Back Issue(s)  

[ ] My new/correct address/phone is listed below

[ ] Enclosed please find a contribution, idea, article or opinion which is submitted for publication in the Pascal News.

[ ] Comments:__________________________
__________________________

ENCLOSED PLEASE FIND:  
$  
£ ____.____

NAME ____________________________
ADDRESS ____________________________
PHONE ____________________________
COMPUTER ____________________________
DATE ____________________________
JOINING PASCAL USER'S GROUP?

- Membership is open to anyone: Particularly the Pascal user, teacher, maintainer, implementor, distributor, or just plain fan.
- Please enclose the proper prepayment (check payable to "Pascal User's Group"); we will not bill you.
- Please do not send us purchase orders; we cannot endure the paper work!
- When you join PUG any time within an academic year: July 1 to June 30, you will receive all issues of Pascal News for that year.
- We produce Pascal News as a means toward the end of promoting Pascal and communicating news of events surrounding Pascal to persons interested in Pascal. We are simply interested in the news ourselves and prefer to share it through Pascal News. We desire to minimize paperwork, because we have other work to do.

- American Region (North and South America): Send $6.00 per year to the address on the reverse side. International telephone: 1-404-252-2600.
- European Region (Europe, North Africa, Western and Central Asia): Join through PUG (UK). Send £4.00 per year to: Pascal Users' Group, c/o Computer Studies Group, Mathematics Department, The University, Southampton S09 5NH, United Kingdom. International telephone: 44-703-559122 x700.
- Australasian Region (Australia, East Asia - incl. Japan): Join through PUG(AUS). Send $8.00 per year to: Pascal Users' Group, c/o Arthur Sale, Department of Information Science, University of Tasmania, Box 252C GPO, Hobart, Tasmania 7001, Australia. International telephone: 61-02-23 0561.

PUG(USA) produces Pascal News and keeps all mailing addresses on a common list. Regional representatives collect memberships from their regions as a service, and they reprint and distribute Pascal News using a proof copy and mailing labels sent from PUG(USA). Persons in the Australasian and European Regions must join through their regional representatives. People in other places can join through PUG(USA).

RENEWING?

- Please renew early (before August) and please write us a line or two to tell us what you are doing with Pascal, and tell us what you think of PUG and Pascal News. Renewing for more than one year saves us time.

ORDERING BACK ISSUES OR EXTRA ISSUES?

- Our unusual policy of automatically sending all issues of Pascal News to anyone who joins within an academic year (July 1 to June 30) means that we eliminate many requests for backissues ahead of time, and we don't have to reprint important information in every issue--especially about Pascal implementations!
- Issues 1 .. 8 (January, 1974 - May 1977) are out of print.
  (A few copies of issue 8 remain at PUG(UK) available for £2 each.)
- Issues 9 .. 12 (September, 1977 - June, 1978) are available from PUG(USA) all for $10.00 and from PUG(AUS) all for $A10.
- Issues 13 .. 16 are available from PUG(UK) all for £6; from PUG(AUS) all for $A10; and from PUG(USA) all for $10.00.
- Extra single copies of new issues (current academic year) are: $3.00 each - PUG(USA); £2 each - PUG(UK); and $A3 each - PUG(AUS).

SENDING MATERIAL FOR PUBLICATION?

- Your experiences with Pascal (teaching and otherwise), ideas, letters, opinions, notices, news, articles, conference announcements, reports, implementation information, applications, etc. are welcome. "All The News That's Fit, We Print." Please send material single-spaced and in camera-ready (use a dark ribbon and lines 18.5 cm wide) form.
- Remember: All letters to us will be printed unless they contain a request
Editor's Contribution

This special issue on the Pascal Validation Suite was prepared primarily by Jenny Nizelinski (now Jenny Sinclair) and Arthur Sale of the Department of Information Science at the University of Tasmania. We owe special thanks to Jenny because she did most of the work for PUG Australasia, and she should have been listed as a "tasks editor" long before now--in fact as far back as 1977!

The Validation Suite represents a valuable weapon in Pascal's arsenal because it provides a common measuring instrument for standards conformance. As has been said before, Pascal now joins a small and elite group of programming languages which has such a collection of test programs.

Note that the Validation Suite is copyright, but that it can be obtained very easily and inexpensively from three (3!) worldwide distributors! ANY user of ANY Pascal compiler ANYWHERE should not pass up this opportunity!

Rick Shaw

Next issue (PN#17) will be Rick Shaw's first issue as editor and is now scheduled for December. (This issue is my last one as editor.) Please don't be alarmed that Rick works for DEC. He will keep PUG and DEC strictly separated. (Besides he is just as funny and crazy a person as I am!) As said in my editorial in PN #15, Rick is a capable administrator (whereas I am not good at delegating responsibility), and he has the luck of being in a nice work environment at DEC's Atlanta Regional Office with ready access to clerical facilities, etc. We were able to recruit section editors (listed on the inside front cover) to whom Rick can now distribute the work of Pascal News. Decentralization is mandatory if Rick is to survive my fate. Good luck, Rick!

About a year and a half ago it would have been hard for me to say goodbye to Pascal News. Now it is really easy! I'm really weary and "burned-out" having worked hard all year--even after having said in issue #12 that I was tired of doing the job. Also I'm comforted that as of about a year ago, it became undeniable obviously that together all of us Pascalers permanently established Pascal as a major programming language--not just "another" language. All progress since then has been and will be pleasant dividends.

Rick's volunteering to be editor for 2 years comes just in the nick of time: a reluctant editor such as myself doesn't contribute to the quality of Pascal News. Rick will provide fresh ideas whereas I'm running out of ideas.

I feel relieved to be rid of the day-to-day responsibility for Pascal News and PUG. (However I intend to help Rick every way I can.) I do admit that working on all phases of PUG and Pascal News has made me a better person. I had the privilege to experience the processes of organizing, accounting, budgeting, editing, filing, printing, archiving, implementing ideas, pasting-up, publishing, planning, mailing, banking, maintaining mailing lists, juggling details, coordinating events, reading faster and writing better, and talking and working and negotiating and learning with other people!

Thanks

It's the honest truth: we've received hundreds of encouraging and favorable comments about Pascal News. It was truly gratifying to receive nice words this year when our hopes were dim and spirits were down. But then it is only appropriate to thank everyone who contributed material and ideas to Pascal News (by sending them in) and made the whole effort possible. Regular contributors were especially valuable. (As an example there is no reader of Pascal News who doesn't know Arthur Sale. He is a Pascal "folk-hero" because his prolific efforts are accompanied with an unforgettable signature and the end-of-the-earth Tasmanian letterhead.) I've been much less an editor than a collector and organizer of information, and I would like to say "Thanks!" and encourage all of you to keep sending in information no matter how small. Unfortunately, I'm sure we still only know less than half of the news concerning the use of Pascal!

We (especially myself) are indebted to the people whose names are listed below. They volunteered their time, energy, and enthusiasm over the last 4 years directly producing and distributing Pascal News (listed chronologically):

- Herb Rubenstein 1977
- Rich Cichelli 1978, 1979
- Scott Bertilson 1978, 1979
- Steve Reissman 1978
- Liz Earl 1978
- Jerry Stearns 1978, 1979
- Kay Holleman 1978
- Tony Gerber 1976, 1977 (Aus.)
- Carroll Morgan 1976, 1977 (Aus.)
- Sara Graffunder 1977, 1978
- Rick Marcus 1979

How do we put together an issue of Pascal News?

Invariably the process begins by catching up (1) on the mail. This means opening an accumulation of what used to be 2-4 weeks worth in the early days of PUG to 2-22 weeks worth recently (I'm talking about trays of mail or 2 meters long!). The mail must be separated (2) into new subscriptions, renewals, inquiries for information, changes of address, incorrect payments (returned), purchase orders without prepayment (returned), miscellaneous queries, and material for publication in Pascal News. (To keep our files uniform and organized we manually fill out an All-Purpose Coupon for new subscriptions and renewals, for requests for old backissues, we manually write out an address label.) The money must be deposited (3) and accounted for (4). New members and renewers must be keyed into the data base (5) and then checked for errors (6). Back issues are mailed (7). The roster increment is run off (8), and the All-Purpose Coupons with tidbit comments are photocopied (9) for the Here and There editor. The material for Pascal News is gathered together in a pile (10) and then sorted (11) into regular categories (Here & There, Open Forum, etc.). The Implementation Notes section is preprocessed (12) (outlined) and given to the Implementation Notes editors. The Books and Articles section is treated in the same way.

The Articles section is planned and received-dates added (13). The Open Forum section is planned (14). At this point all parts of the issue are attacked (15) at the same time including the sections (6) and (11) just noted. The Open Forum (16) which delays actual page layout and pastetup of the rest of the issue.

When the camera-ready copy of the editorial and everything else is ready (or nearly so), paste-up with rubber cement on large computer-listing paper begins (18). Each sheet of large paper was previously titled and page-numbered (19) to produce the 2 copies for PUG(UK) and PUG(AUS) to print from. It is then sent (20) to the printer together with a print order.

Unfortunately, these events don’t always occur in this order, thus creating synchronization problems. Needless to say we are always alert for news about Pascal in other journals and from people who call on the phone.
Editor's Contribution

In Closing

As an escape clause, I've always listed: "as well as the ideas behind Pascal" together with "promoting the use of Pascal" as a purpose of Pascal User's Group. We all know that Pascal is not a perfect language, but that it best embodies the ideas of the structured-programming revolution of the 1970's.

Acceptance in the United States has been the icing on the cake and was crucial to Pascal's success as a popular programming language (sorry, ALGOL-68!). So if you stop to think, it is important to note that the Pascal movement in the United States was spearheaded by George Richmond (with some initial help by Lyle Smith, a friend of Niklaus Wirth) at the University of Colorado Computer Center primarily during the years 1972 to 1975.

The effort has been continued by Pascal User's Group via Pascal News by communicating "vast quantities of information" from late 1975 to present. Pascal News and Pascal User's Group (that is, all of us!) succeeded in centralizing authority for Pascal's acceptance, development, and standardization.

What we have done through the medium of Pascal News which was not being done (and probably could not have been done) by any other journal was to openly advocate the superiority of the principles behind Pascal. Perhaps we succeeded in shaking up enough people to accelerate rationality in programming and sensibility in computing by a number of years.

We oversaw a political process and interjected some self-fulfilling prophesies to keep the action rolling. Inevitably we all were affected by the spectacular outcome ourselves!

- 1979/10/21.

DON'T FORGET TO RENEW your PASCAL NEWS Subscription.

October 23, 1979

PUG is not dead!

The first question you all probably asked yourself when you heard the news is: "How will it change?" Well, the answer is: "Not much at all!". It is going to be the same old PUG you grew to love and respect. With the same editorial policy, and the same informal approach to publication. (But it will come out four times a year, that's my only promise!) The only noticeable change will be the Editor and who does all the work. I am not a human dynamo like Andy, so I have had to enlist the aid of many volunteers (see Andy's column) to help me with all the work that used to be done almost single handed. If the next few issues are not as slick as you are used to from PASCAL NEWS, please bear with us. It may take one or two issues to get it right. Number 17 will be out by the end of the year and we hope to have Number 18 published by the end of February.

As a closing note, I would like to ask that all of you start using the new All Purpose Coupon published in this and subsequent issues of the News. It will speed the transition up tremendously between Andy and myself.

See you all next issue. Long live PUG!

Rick Shaw

DIGITAL EQUIPMENT CORPORATION, 5775 PEACHTREE-DUNWOODY ROAD, SUITE 100D, ATLANTA, GEORGIA 30342
(404) 252-2600
Introduction to Special Issue

The Pascal Validation Suite

Aims and Methods

by A.H.J. Sale,
Department of Information Science, University of Tasmania.

1979 July 13

Once upon a whisper-time - so it is said but who would believe it? - long before the Minnipins reached the Land Between the Mountains, the Glacken of Then played upon his bells and the beetle-borses, which had come to infest the countryside, fell upon their backs and waggled their legs for a space and died. But the smell of dead beetle-borses was great, and the bells could do nothing about that, and this I believe.

Pretend-story, told by Glacken to Glacken to Glacken, from Then to Now. (by Carol Kendall - The Whisper of Glacken)

1. Definition

A Pascal validation suite? What is that? Ignoring the facetious definitions, such as a suite of motel rooms where a Pascal salesman wines and dines clients and promotes the features of the language, there are still a lot of possibilities.

It might be a set of programs that check whether some other (input) test is a valid Pascal program or not. (This may run into the halting problem, well-known to be intractable). But it isn't. The shortest definition I can supply which describes the validation suite I am talking about is:

The validation suite provides programs and procedures whereby the correctness of an implementation of a Pascal processor may be tested.

2. Syntax + Semantics

There are two key words in this definition which have been carefully chosen, and which deserve consideration. Firstly, the definition encompasses a Pascal processor, not a compiler. The definition therefore covers processors that compile native machine-code and run it, processors that utilize an intermediate code and an interpreter (for example the P-compiler), direct execution systems and pure interpreters.

But more importantly, the term processor encompasses both the analysis of the source text of a program on an execution system. I am not interested solely in determining that a compiler is "correct", whatever that means, but in determining that the compiler-machine pair is correct. To take a very simple example, the following program fails on some processors:

```pascal
b := true; c := false;
if (b = (not c)) then writeln('PASS');
else writeln('FAIL');
```

What goes wrong? On analysis of the failures, the compiler seems to generate good, correct, code. Giving P-like code as an example, the test compiles to:

```pascal
LOAD b {to stack}
LOAD c {to stack}
NOT {logical inversion of c}
EQUAL {test equality, leave logical result}
BRFL ... {branch if false}
```

The flow usually resolves into a fail of the compiler assumptions. The NOT instruction perhaps does not do the transform `true = false`, but does a whole-word bit inversion. Coupled with the action of an equality test, of course it may then fail! The bit pattern resulting from inverting the false-pattern is not necessarily the same as the true-pattern.

Resolution of this problem can take several paths, presuming that the machine architecture is fixed. (This assumption is false for interpreters, of course). The simplest resolution is to ensure that every occurrence of the not-operator results in code that allows only the bit-patterns 00...00 and 00...01 to be generated, as is the usual representation of boolean values in more than one bit. In the Burroughs B6700 this can be achieved by omitting the instructions:

```pascal
LNOT {workaround logical inversion}
ISOL(0,1) {isolate the 0-th bit}
```

Another resolution hinges on the manner in which the branch tests are done. Suppose that the machine, when faced with a complex logical expression, generates a sequence of branch instructions which lead eventually to loading a true or false value (or a branch if that is required.)

A third resolution permits other representations for true and false (perhaps including don't care bit positions), but realizes that an unusual representation may require special coding

(a) at conditional branch points

(b) wherever ordering is important (eg. for parameters of ord, succ and pred applied to boolean operands), and

(c) in 'relational expressions' involving booleans, which may be turned into something other than number comparisons (eg. exclusive-or).

For example, in the Burroughs B6700 the representations ??...?? and ??...?I would suffice as the conditional branch instructions sense only the right-end bit, and consequently complex code would only be needed for the relatively unusual case where the ordering of the values was relevant. But this is not a treatise on implementing boolean.

Suffice it to note that testing a processor involves the semantics as well as the syntax, and the machine as well as the compiler.

3. Ultimate futility or useful weapon

The other carefully chosen word in the definition is correctness. Many people have pointed out that testing cannot prove a program to be correct; it can only uncover bugs in it. Since a compiler is a program, and only part of a Pascal processor, is testing therefore an exercise in futility that we should abandon in favour of proving the compiler-machine pair to be a correct implementation of Pascal?
Of course, yes, if we were capable of it. But proof procedures are still human processes, and still subject to error and oversight. Therefore, even proved systems should be subject to testing in order to uncover weaknesses of the proof or oversights. It will be obvious that this is necessary when it is realized that Pascal-P, despite its heritage of careful design, its origin in Zurich, and its extensive testing, still has errors in it which can be detected in most of its descendants.

The validation suite is a set of programs, methodically assembled (and otherwise) which therefore exercise a Pascal processor fairly thoroughly and hopefully uncover many of the flaws in its design or mistakes and deviations from the intended actions.

So much for the theory. How does it work in practice?

4. Conformity

The most obvious set of tests to incorporate are those set down in the Pascal Standard as required of a Pascal processor, or implied to be. These were generated by systematically working through the Standard and wherever it said something was allowed, writing a program to check that it was. All such programs are, of course, standard Pascal.

Since the program should execute, it is arranged to print 'PASS' if it works correctly. Some such tests check semantic details, and incorporate run-time checks that lead to failure messages. By the nature of the test, several conformity checks can be included in one test program. Any failure is sufficient to show up a flaw: successful programs exercise all tested features.

However there are three problems: iteration, depth and irregularity.

The first arises with iterative/recursive syntax, or axiomatic recursive semantics. Clearly no program can test all cases of a potentially infinite system! This is tackled by the "one, two, many" principle, named after the primitive enumeration systems of nomad Bushmen in southern Africa. The test will include a minimum case of the construct, probably its successor, and a case which is a small multiple of iterations/recursions, and quite plausible. For example, for identifiers in a while declaration:

```pascal
program txxx(output);
var
  onlyone : integer;
  first,second:boolean;
  x0,x1,x2,x3,x4 : char;
begin
  ...
end
```

Since no processor in a finite computer can provide for infinite recursion/iteration, these tests establish a *prima facie* case that the construct is present and works for small instances up to some unknown limit. To establish that the limit is sufficiently high that it will almost never prove to be a problem - what I have called a virtual infinity - another program is written which has a repetition slightly (and only slightly) beyond a plausible maximum. Perhaps 100 cases might be enough for some things, and 20 for other things. Such tests are regarded as not being compulsory, and form part of the quality measurement category. (Nevertheless, a processor may fail even a quality test by getting itself knotted, or giving an erroneous diagnostic, or behaving in an unexpected fashion...)

5. Deviance

Besides saying what ought to be allowed, the Pascal Standard says what ought not to be allowed, both explicitly and implicitly. If a processor allows such constructs, it is either a deviant processor and ought to be fixed, or it embodies some deliberate extension which ought to be documented.

Note that it is pointless trying to detect all possible extensions: they form an infinite set! In any case this is not the purpose of deviance tests. Their purpose is to detect the many traps and failures to enforce reasonable restrictions which compiler implementors can unwittingly place in the way of users.

Some deviance tests are directly suggested by the Pascal Standard. Good examples are afforded by the restrictions placed on for-statement control-variables. But by far the larger group are not. These rely on the test program writers' intelligence and knowledge of compilers and on the ability or experience to recognize possible problem areas. Two examples will illustrate this.

The first was derived from some experience with an optimizing compiler which generated different code for

```pascal
j div k
```

depending on whether \( k \) was a constant power of two or not, or so the documentation said. This immediately challenged the hardware designer in me to check this out, and challenged my software designing side to prove it correct. Since the optimization relied on a property of the integer representation, it was potentially possible for it to fail. Sure enough, on this compiler, it did: \( j \div k \) returned different results for the same value of \( k \), solely depending on whether it was a constant or not.

The third problem is irregularity. Successful execution of a test is no guarantee that the same feature will be handled correctly in a different context. Unfortunately there is no way to know all possible context changes that might affect the outcome, and the designers of the test programs have had to draw on their knowledge of machines and implementation techniques to explore this difficult area. A good example is the implementation of type boolean. Clearly, from our earlier discussion, relational operators must be tested separately for booleans and other scalar types.

Another example which maybe should be in the test package, but isn't, is the following

```pascal
csquared:=2.0*alpha*beta;
if csquared = a*n*b*b then
  writeln(a,b, sqrt(csquared));
```

A particular processor noted that `csquared` was used immediately after its assignment and modified the code to leave this value on the runtime stack to eliminate the fetch. A separate optimizing routine realized that the if-statement had no effect once the debug print was factored out, and deleted the whole of the if-statement code. Result: the stack grew every time this code was executed... Such interactions require cunning and experience to deduce, let alone devise tests for their presence.
The second case grew out of a knowledge of the workings of the compatibility algorithm of most Pascal compilers, and some strict wording in the Standard. Strictly

```pascal
type
digitstring = packed array[1..10] of '0'..'9';
is not a 'string' type, and not compatible with the usual quoted string constant. Thus

```var
d : digitstring;
begin
  d := '0123456789';
end
```is not correct Pascal: it is a deviation or an extension. Of course, compilers that allow it as an extension ought to do it properly (i.e. consistently with the axioms of Pascal), so this naturally gives rise to the attempt:

```pascal
d := 'ABCDEFGHIJ';
```which is plain impossible.

Enough of examples. It will be obvious that every deviance test program is non-standard, and that they should mostly fail to execute to completion. Unlike conformance tests this means that each deviance test tries one and only one deviation. Otherwise one failure might mask another, more serious. Conformance and deviance tests make up the majority of the tests (71% of the current suite), and do the bulk of the exercising of the compiler.

6. Errors and implementation features

The Pascal Standard also specifies a number of situations which are specified to be errors, implementation-dependent, or implementation-defined. Each of these gives rise to a test or tests that evoke the feature so specified.

In the case of errors and implementation-dependency, the purpose is to enable documentation of the error-handling capability of the processor. Hopefully all errors are detected; in practice some of them survive into production software and may even become part of programmer's assumption kit... All such programs are directly suggested by the Standard, evoke one and only one error, and are standard Pascal except for the error.

Naturally, implementation-defined features should be documented, and the corresponding tests attempt to do this by a variety of techniques. Some, like printing the value of maxint, are simple standard Pascal. Others, like trying to detect the significance limit of identifiers, rely on assumptions about the nature of the lexical analysis and scope and are clearly not standard. The inclusion of these tests is really to improve the documentation of Pascal processors.

The number of tests in those categories is small and unlikely to change much. The specific things they test are more or less fixed by the Standard.

7. Quality

All the preceding tests address the problem of whether the processor conforms to the Pascal Standard and, where necessary, how. It is also relevant to ask whether the implementation measures up to some quality standards, and the validation suite therefore contains tests which attempt to assess quality, however you define it.
8. Feedback

No validation suite can ever be perfect. Since its task is infinite, one can only approach perfection with varying degrees of success. Consequently, it is very important that a continuous revision of the suite is maintained, with three main objectives in mind:

(a) Removal or modification of tests that do not agree with requirements of the Pascal Standard.
(b) Addition of new tests arising from experience and context changes, and therefore unexpected interpretations.
(c) Expansion of quality control tests aimed at improving the quality of Pascal processors in all dimensions of choice.

Suggestions for categories (b) and (c) are therefore welcome. Complaints about category (a) are even more important. All will be read, digested, and carefully considered, though clearly some purely idiosyncratic tests may not make it into the package.

There is no formal bug-reporting service. Simply write to me at the following addresses:

up to January 1980
Professor A.H.J. Sale,
c/- Professor D.W. Barron,
Department of Computer Studies,
The University,
Southampton, England SO9 5NH
UNITED KINGDOM.

from January 1980
Professor A.H.J. Sale,
Department of Information Science,
University of Tasmania,
GPO Box 252C
Hobart, Tasmania 7001
AUSTRALIA.

Please enclose a listing and any other information relative to the report/request.

But this is not enough! Simply casting the validation suite out into the wide world calls out for more. My own curiosity is reason enough, but in addition it has become apparent that overseeing the results of other people's passes of the validation suite against a processor can be extremely valuable.

Accordingly I ask you to write to me if you are prepared to send documentation on a validation run for a processor I haven't already collected. I will let you know if we already have data on that system.

The documentation I would want is:

(a) a complete set of listings of runs of the validation programs, annotated to explain any obscure effects.
(b) a validation report, similar to the one produced for our own compiler.
(c) an accurate identification of the processor (compiler source and date of acquisition or version, machine identification). A manual would be useful too.

With help we can begin to assemble a comparative list of processor performance, and watch the way the situation evolves.

9. Utilization

I hope to see copies of validation reports for processors of significant interest in future editions of Pascal News. Obviously implementors will want to fix minor bugs, but may baulk at fixing difficult ones or publishing the results. Some comparative results published by responsible users will assist readers of the News to assess comparative merits of compilers, may bring some collective user pressure to bear on the implementors/maintainers to fix even the more persistent and difficult bugs, and will assist the Users Group to assess which systems are still active or most in use. I hope to see the validation suite being used by at least two groups of people: the implementors/maintainers and the users.

Implementors of course will use the package to check out their product; maintainers would be well-advised to do the same after each major revision. They will also be able to compare their efforts against other compilers more easily, and a bit more competition will be good for both suppliers and users.

Users will be able to use the package to bring pressure on suppliers to conform to the draft standard, and therefore use the package to evaluate the relative merit of two or more systems. Also, because the validation suite is virtually the Pascal Standard cast into test programs (with some reservations), access to the essential concepts of the Pascal Standard will be made more clear by inspection of the conformance and deviance tests.

Naturally, as the draft Pascal Standard stabilizes, the agreed resolutions will be embodied in further tests, thereby both providing an enforcement technique, and a method of publicizing the agreement. For example, this may happen in the area of the pack and unpack procedures which are perhaps overly restrictively defined.

10. A Confession

Once upon a time, long ago, programmers who could write tricky code were prized. This pleased them, because they were being well-rewarded for doing what they enjoyed: creating private masterpieces of complexity on a small scale. Any modern abstract artist would understand the feeling.

Now, programming has completely changed. We strive to write correct programs together with their proofs. The creativity, and the accompanying surge of pleasurable sensations on completion of a particularly difficult task, are still with us, but in a different form, and with different nuances and vibrations. I enjoy this tremendously when administrative chores allow me to indulge myself in writing programs, and I wouldn't give up the advances we've made in programming methodology for anything.

And yet, there is still that fascination with those tricky, nasty jewels we created. I must therefore confess that I believe that when the last tricky programmer on earth expires his or her last breath over a soggy sheet of paper, he or she will probably be writing a validation program. It seems to be the last refuge left for contorted thought, for to paraphrase a famous fictional detective (I think it was Hercule Poirot), how can you discover what crime has been done unless you can put yourself into the frame of mind of the criminal?
11. Some favourite tests

I must confess to having some favourite tests, which either are utterly useless and indulge a peculiar sense of humour, or are particularly devastating. I share them with you in case you might, too, share this sense of humour.

(a) Syntactic
Test 6.4.2.3-1
The interesting thing here is the peculiar scalar type

```
singularitytype = (me);
```

which doesn't seem to be useful for anything, though you can assign to it, test it, etc. The apparently similar (test 6.4.1-1): type

```
purelink = ′{purelink}′;
```

is less amusing because it does have at least two distinct potential uses.

(b) Context-sensitive
Test 6.4.3.3-4
Here the use of a field-name which is already defined trips up a number of compilers. They don't wait to find the colon before rushing into analysis. Most Pascal-P compilers inherit this one.

(c) Scope
Test 6.2.2-2
The sheer perversity of being able to write

```
if true = false then writeln (′PASS′)
```

is delightful.

(d) Execution
Test 6.8.3.9-7
The beauty of this one lies in two aspects: many processors completely fail it, and our compiler turned out to pass it quite unexpectedly. It comes as a surprise to devise a test that you confidently expect to fail on your own implementation, and then the thing makes a fool of you by working.... And then you have to work out how it outsmarted you. To add to its perverse charm, it usually turns out to have a simple resolution which, though unattractive, requires only that maxint be reduced by one.

(e) Error handling
Test 6.6.5.2-6
I must admit that the attraction of this one lies both in the weird variety of effects it can evoke, and to some extent by the clarity with which the aliasing problem points an accusing finger at the file buffer concept. Interestingly, a simple restriction would remove the problem, by simply not permitting file-buffers in such contexts, but it would introduce more irregularity....

Acknowledgements

I wish to acknowledge the great debt the Validation Suite owes to a multitude of people. Those I single out for special mention here contributed especially, but there are many other contributors for which there is not sufficient space.

Brian Wichmann: for initiating the project, for carrying it out throughout the first phase, and for many insights and tests, and as joint author.

Andy Mickel: for encouragement to continue, despite over-runs.

R.D. Tennent: for many critical comments on semantics.

Roy Freak: for patient and hard work in assembling around 300 programs to a consistent style and putting up with my nit-picking and niggling.

Nigel Saville: for diligence in interpreting often partial instructions and creating a large number of provably correct (or provably incorrect) programs in the rewriting phase.

Jenny Mizielinski: for laboriously and carefully typing (seemingly endlessly) abtruse documents full of mysterious numbers, each of which was highly significant.

The Sale Family: for putting up with grunts and groans, and with listings strewn around the sitting room.

Of course, to all my correspondents who contributed unknowingly must go a special kind of thanks. Without your stimulation, the Validation Suite might not have had the firm basis it now has.

Arthur Sale
Tasmania
1979 August
The Pascal Validation Suite
Version 2.2

Revision History

Version 2.0 was the first release of a completely rewritten package which
was based on earlier work by B.A. Wichmann and A.H.J. Sale. This earlier
work is considered to be version 1, and is now obsolete.

Version 2.0 was indexed to Working Draft 3 of the Pascal Standard as published
in Pascal News #14. Version 2.1 fixed up a few bugs detected after release.
Version 2.2 is altered in indexing to refer to the draft ISO Standard
document ISO/TC97/SC5/N462, and incorporates more tests which facilitate
the timing and measurement of quality in Pascal processors. Subsequent
revisions will be issued when either detected errors in the package require
a revision, or when a new version of the draft standard is released.

Purpose

The validation suite is provided to exercise Pascal processors and to
determine by testing whether the processor conforms to the requirements of
the Pascal Standard or not, and to provide a common set of programs for
documenting implementation-dependencies and quality of implementation. It
is strongly oriented around the draft Pascal Standard, and tests are
suggested by that document. A few proposed tests have been omitted because
it has been suggested that the draft Standard will be revised in that area,
but these are few. It follows therefore that any revisions of the draft
Standard will cause revisions of the validation suite. The suite currently
contains over 300 programs.

Acquisition

To acquire a machine-readable copy of the validation suite, apply to one of
the distribution centres. It will be necessary to fill in a software licence,
and a small fee (around US$50) is charged to cover costs. The fee covers
the supply of a magnetic tape, the copying of the validation suite onto the
tape, airmail postage to the address provided, and a limited notification
service relating to later releases or inaccuracies detected in the suite.

Restrictions

The conditions of release prohibit the distribution of the package to third
parties so as to limit the growth of unauthorized and inaccurate versions.
The likely incidence of change if the draft standard is revised will show
the desirability of this requirement. However, no restriction is placed on
the use of the package for validating Pascal processors, for benchmarking,
for acceptance tests, for preparing comparative reports, and similar
activities, nor on the distribution of the results of such use.

The validation suite is expected to be widely used and distributed, not
restricted to a small subset of the user community.

Feedback

No special reporting mechanisms have been set up. However, the authors
will attempt to produce revisions of the validation suite and distribute
them to the distribution centres as necessary, and to publicize their
availability.

Information relating to a pass of the validation programs against a
particular processor would be welcome at the University of Tasmania, but
it must be understood that the authors cannot provide full reports on all
listings provided.

Of particular value would be any tests that produced entirely unexpected
results which may be of wider interest, or which are without any reasonable
explanation from the user's point of view. In some cases the authors may
be able to deduce the likely cause, or recognize an epidemic of common
flaws. Any correspondence which points out an error in the classification
of the programs in the suite, or in its construction, or suggests a new
test, would be most welcome.

Address for suggestions or complaints:

September '79 - January '80:
Professor A.H.J. Sale,
c/- Department of Computer Studies,
The University,
Southampton, England  S09 5NH
UNITED KINGDOM.

February '80 onwards:
Professor A.H.J. Sale,
Department of Information Science,
University of Tasmania,
GPO Box 252C,
Hobart, Tasmania 7001
AUSTRALIA.
**Distribution Format and Addresses**

Character set

```
!"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
```

**THIS TAPE CONTAINS 4 FILES:**

**FILE 1:**
- The character set used in the validation suite and details of the tape structure.

**FILE 2:**
- The skeleton program which identifies each test program.

**FILE 3:**
- The documentation of the Pascal processor validation suite.

**FILE 4:**
- The suite of Pascal test programs.

**Addresses:**

**DISTRIBUTION IN AUSTRALIA, NEW ZEALAND AND JAPAN:**

PASCAL SUPPORT,
DEPARTMENT OF INFORMATION SCIENCE,
UNIVERSITY OF TASMANIA,
BOX 2520, G.P.O.,
HOBART 7000
TASMANIA
AUSTRALIA.
61-62-230561 EXT 435
DISTRIBUTION COST: $A50
FOR DISTRIBUTION IN NORTH AMERICA -

RICHARD J. CICHELLI,
C/- ANPA RESEARCH INSTITUTE
BOX 598
EASTON, PA. 18042
U.S.A.
(215) 253-6155
DISTRIBUTION COST: $US50
FOR DISTRIBUTION IN EUROPE -

DR. B. WICLMANN,
NATIONAL PHYSICS LABORATORY,
TEEDINGTON, TW11 0LW
MIDDLESEX,
ENGLAND
01-977 3222 EXT 3976
DISTRIBUTION COST: NOT KNOWN

---

(C) COPYRIGHT
A.N.J. SALE
R.A. FREAK
JULY 1979

ALL RIGHTS RESERVED

THIS MATERIAL MAY NOT BE REPRODUCED OR COPIED IN WHOLE OR PART WITHOUT WRITTEN PERMISSION FROM THE AUTHORS.

DEPARTMENT OF INFORMATION SCIENCE
UNIVERSITY OF TASMANIA
BOX 2520, G.P.O.,
HOBART 7001
TASMANIA
AUSTRALIA

---
A Pascal Processor Validation Suite

by
B.A. Wichmann
National Physical Laboratory
Teddington, Middlesex,
TW11 0LW, United Kingdom

A.H.J. Sale
Department of Information Science
University of Tasmania,
GPO Box 252C,
HOBART, Tasmania 7001

Abstract

The document describes 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.

NOTE

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.

Dated: 20 August, 1979

Version: 2.2

1. INTRODUCTION AND PURPOSE

This paper describes a suite of test programs which has been designed to support the draft Standard (Addyman, 1979) for the programming language Pascal (Jensen & Wirth, 1975) prepared for approval by ISO. (In the rest of this paper, the draft Standard is simply referred to as the Standard). It therefore follows similar work done by AFSC (1970) for COBOL, and Wichmann (1973,76,77) for Algol 60.

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 interpretable 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 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-validation of a processor after modification to incorporate a new feature, or to fix 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 use in influencing 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.

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

(ii) 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.

(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.

(v)* The lexical tokens used are in conformance with the conventions set out in the draft ISO standard, and reproduced in an appendix. 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.)

(vi) 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.

(vii)* The program writes to the default file output, which is therefore declared in the program heading. (Note: one test - the minimal program - does no printing; a few cross-references are virtually the same).
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 ")" directly juxtaposed in this way. The syntax of a header comment in EBNF 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 tspspd10. This technique may also be used to name a program source text file name 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 at 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 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 some cases the program will print a message beginning "DEVIATES", and users of the tests must distinguish between genuine 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 past flaws detected. 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 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. The tests include the following, amongst others:

- 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

A final category is 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, CLASS=EXTENSION, SUBCLASS=CONFORMANCE}

The subclass gives 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
      final file of tests.
   3. A copy of this document in machine readable form.
   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.

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** (host, computer, origin of processor,
  version):
- **TEST CONDITIONS** (tester, date, test version):
- **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 extension:
    - ...
  - Details of deviations:
    - ...
- **ERROR-HANDLING**
  - Number of errors correctly detected = ?
  - Number of errors not detected = ?
  - Details of errors not detected:
    - ...
- **IMPLEMENTATION DEFINED**
  - 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**
  - Number of tests run = ?
  - Extension present = ?
    - as above for this extension.
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 (1973) and A.H.J. Sale (1978) and significant contributions have also been made by A.M. Addyman and R.D. Tennent. R. Freak and N. Saville 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


Appendix: Documentation of Skeleton Program

Purpose
The skeleton program provided is a standard-conforming Pascal program that will identify each test in a file of test programs. It is distributed with five stubs which are intended for user modification so that the program will serve as the parent of an automatic system for running the tests. With the large number of tests in the validation suite, such a system is important. As distributed the stubs simply print the text of the test program.

Implementation
If the recipient's Pascal processor will accept upper- and lower-case letters, and the ISO standard lexical representation, then the program should be able to run directly. If lexical substitutions are necessary, these should be made. No lines approach the limiting length of 72 closely, so some expansion room is possible. If only one case of letters is available for the source text, then the whole program should be upper-cased with the exception of the character constants in the main program and the procedure convert. If however, the Pascal processor is also limited to upper-case characters in the char type, these too will have to be converted, and the whole of the validation suite will have to be converted to upper-case alone before the program will run.

Modification
Once the recipient has verified that he has the skeleton working correctly, it can be modified to other purposes. If the only use desired is the construction of many individual files, each with a single test program, then newSuite and endSuite may be made dummy procedures. Procedure newProgram can open (rewrite) a file with the converted name supplied, which is guaranteed unique, leaving processLine to write the text to this file and endProgram to do any necessary closing of the file.

Alternatively, the program may be modified to construct a job deck including job control statements. The newProgram procedure will have some more complex actions to take, and the endProgram procedure will initiate the job into the operating system queues. In the event that the user wants to batch up the tests in lots of 20 or so, the access to the variable count allows newProgram and endProgram to take appropriate action every twentieth test. Or again, if the whole lot is to be batched or submitted as a simulated time-sharing job, the newSuite and endSuite programs can be used to initialize and initiate.

Additionally, the newProgram procedure is given access to the test program name and its class. Specially tailored programs can be written to only initiate conformance tests, or only conformance tests relevant to section 6.5 of the Standard, or whatever the user desires. If necessary, specifications could be read in, most easily by a modified version of the newSuite procedure.

System documentation
The main program alternates between dormant states where it searches for the header comment starting a test, and active states where it processes lines keeping an eye out for the closing end of the test. When it finds a header comment, it extracts the data for passing to the stub procedures newProgram and endProgram.

The procedure extract does this extraction from the line buffer. It essentially assumes correctness of the header syntax. The program name is stored in a special record.

The procedure convert is not necessary to the distributed version apart from a demonstrative use. It is provided to convert test program numbers from numeric format (e.g. 6.2.8-4) to an alphabetic format suitable as an identifier or file name (e.g. t6p2p8d4). It switches the format in the name record from whatever it is to the alternative format. The format as supplied to newProgram is digitized.

The procedure readLine does simply that. The line is stored in an internal buffer line.
The procedures `newprogram` and `endprogram` have value parameters with exactly the same name as their global counterparts. This is to hide the global variables from them and provide some measure of robustness against errors. The `status` parameter of `newprogram` is listed explicitly to emphasize the possible change of value.

There are few features of Pascal in the program that may not be implemented. There are two goto statements leading to a disaster-exit label; if non-local gotos are not implemented the gotos may be omitted or replaced by a `halt` equivalent. The packing in `charvectype` is only provided to allow comparisons with character array constants ("strings"). All identifiers have been checked for uniqueness over the first eight characters, and no non-standard usages have been detected in the program. If it is necessary to alter the program to recognize header comments by '(*T' instead of '{T', some changes will be necessary on two lines of the main program.

**Lexical structure of procedures**

```pascal
main
  readaline
  convert
  extract
    scan
  newsuite*
  endsuite*
  newprogram*
  endprogram*
  procline*
```

**Notes:**

* user-modifiable stub procedures.
† a call structure diagram is also provided overleaf.

**Distributed version**

As distributed the skeleton program prints all the test programs, one per page, in sequence. The headings simply illustrate some features of the package. Warning: this takes a lot of paper, so be prepared.

**Date of documentation**

1979 July 13

**Author**

A.H.J. Sale, University of Tasmania
This skeleton program is provided to process the file of test programs in the validation suite. As supplied, it simply lists the test programs in a suitable format, but it is provided with stubs which can be modified by users of the suite to select individual test programs, classes of programs, or particular sub-classes, and to write these to named files or initiate them in a job stream.

The particular stubs of interest are:

- newsuite
- newprogram
- proceseline
- endprogram
- endsuite

As supplied these are practically only dummy routines.

This program is written in Standard Pascal according to the ISO Standard. It should compile and run without error if its lexical representation is acceptable to your processor.

(C) Copyright 1979 A.H.J. Sale, University of Tasmania.

program skeleton(input,output);
label 999; { used for disaster exits }

const
maxnamesize= 20;
{ size of field in name record }
nameoverflow= 21;
{ maxnamesize + 1 }
maxlinesize= 72;
{ size of line array }
lineoverflow= 73;
{ maxlinesize + 1 }

type
namesize= 1..maxnamesize;
linesize= 1..maxlinesize;
statustype= (dormant,active,terminated);
{ Is program being processed? }
charvectype= packed array[namesize] of char;
{ names }

record
charvec: charvectype;
length: 0..maxnamesize;
version: (digitized,alphabetized)
end;
{ used to hold program names }
natural= 0..maxint;
{ very common type }
linetype= array[linesize] of char;
{ used for line buffers }
loopcontrol= (scanning,found,notfound);
{ for controlling scan loops }
classtype= (conformance,deviance,implementationdefined,
errorhandling,quality,other);
{ category of test }
var
name: nametype;
{ name of current program }
class: classtype;
{ category of current program }
status: statustype;
{ current status }
count: natural;
{ program sequence number }
line: linetype;
{ the line buffer }
linelength: 0..maxlinesize;
{ holds actual number of chars in line }
procedure readaline;
{ Reads an input line }
var
i: 0..lineoverflow;
ch: char;
begin
if eof(input) then begin
  writeln(output); { ***ERROR IN READALINE - HIT EOF' }
goto 999
end else begin
  i:=0;
  while not eoln(input) do begin
    i:=i+1;
    if (i > maxlinesize) then begin
      writeln(output); { ***ERROR IN READALINE, LONG LINE' }
goto 999
    end;
    read(input,line[i]);
  end;
  read(input,ch); { disposing of the line marker }
  linelength:=i;
  { Textfiles must have eoln before eof }
end; { of procedure readaline }

procedure convert(var name:nametype);
{ This procedure exchanges the representation of the name between digitized (eg 6.2-1) and alphabetized (eg t6p2d1).
  It inspects the version code and reverses it. }
var
i: linesize;
ch: char;
begin
if (name.version = digitized) then begin
  { We need to alphabetize the name }
  { Extract assures that digitized length never exceeds maxnamesize-1, so leaving one char expansion space. }
  for i:= name.length downto 1 do begin
    case name.charvec[i] of
      '.': ch:='p';
      '-': ch:='d';
      '0','1','2','3','4','5','6','7','8','9':
        ch:=name.charvec[i]
    end;
    name.charvec[i+1]:=ch
  end;
  name.charvec[1]:='t';
  { Name is converted }
  name.length:=name.length+1;
  name.version:=alphabetized
end else begin { version = alphabetized }
  { We need to digitize the name }
  for i:=2 to name.length do begin
    case name.charvec[i] of
      'p': ch:='.';
      'd': ch:='-
      '0','1','2','3','4','5','6','7','8','9':
        ch:=name.charvec[i]
    end;
    name.charvec[i-1]:=ch
  end;
  name.charvec[name.length]:=' '; { Name is converted }
  name.length:=name.length-1;
  name.version:=digitized
end { of if }
end; { of procedure convert }
procedure extract;
{
This procedure extracts the program-number and class from the
header comment of a test program. It sets global parameters
name and class. There is some simple error-handling in the
procedure, but it is generally assumed that the header
comments conform to the syntax. The error-handling is just
in case.}

var
1, { used to scan the line }
isave: 1.lineoverflow; { used to save value of 1 }

lengthofname, { holds length of name found }
k: namesize; { used to fill name }

begin { of extract }
{
Start scanning at 1 even though we know it begins '{T'}
}
i:=1;
{ Scan until we find a digit }
scan('0', '9');
{ Now assured of a digit so save the index, and look for
the closing comma }
isave:=1;
{ In case spaces between number and comma }
while (line[i-1] = ' ') do i:=i-1;

{ Fill in the discovered name }
lengthofname:=isave;
if (lengthofname > maxnamesize) then begin
lengthofname:=maxnamesize-1
end;
for k:=1 to lengthofname do begin
name.charvec[k]:=line[isave+k-1]
end;

{ Space-fill so as to allow string comparisons }
for k:=(lengthofname+1) to maxnamesize do begin
name.charvec[k):=''
end;

{ Adapt fill in the rest of the record }
name.length:=lengthofname;
name.version:=digitized;

{ Now scan for the '=' sign that precedes the class }
scan('=','=');
{ And an alphabetic character following. The test may
also let through some non-alphabetics, but no matter. }
scan('A','Z');
{ Identify the class by its first letter. It is always
upper-case. }
if (line[i] = 'C') then class:=conformance
else if (line[i] = 'D') then class:=deviance
else if (line[i] = 'E') then class:=errorhandling
else if (line[i] = 'I') then class:=implementationdefined
else if (line[i] = 'Q') then class:=quality
else begin
{ Error, not recognized }
writeln(output);
writeln(output,'*****ERROR IN EXTRACT - WHAT CLASS?');
class:=other { default }
end;

{ Now we have established the desired values, so return }
end;
{ of procedure extract }

APPENDIX 1
PAGE A1-5

procedure scan(lowch,highch:char);
{
Scan moves the index i along the line until it finds
a character lying between lowch and highch inclusive.
It includes some simple error-handling which terminates
the program. }

var
state : loopcontrol;

begin { of scan }
{
Set loop to scan forwards }
state:=scanning;
{ Loop invariant R1 =
"characters from line[initial i] to line[i-1] are
not in the desired subrange." }
while (state = scanning) do begin
if (i > linelength) then begin
{ No more to go, so get out }
state:=notfound
end else if (line[i] >= lowch) and (line[i] <= highch)
then begin
state:=found
end else begin { Char is not in range }
i:=i+1
end; { of if }
end; { of while }

if (state = notfound) then begin
writeln(output);
writeln(output,'*****ERROR IN SCAN - REACHED LINE END');
goto 999 { in outer block and give up }
end;

{ Return, leaving i at found character }
end; { of procedure scan }

APPENDIX 1
PAGE A1-6

procedure extract;
{
This procedure extracts the program-number and class from the
header comment of a test program. It sets global parameters
name and class. There is some simple error-handling in the
procedure, but it is generally assumed that the header
comments conform to the syntax. The error-handling is just
in case.}

var
1, { used to scan the line }
isave: 1.lineoverflow; { used to save value of 1 }

lengthofname, { holds length of name found }
k: namesize; { used to fill name }

begin { of extract }
{
Start scanning at 1 even though we know it begins '{T'}
}
i:=1;
{ Scan until we find a digit }
scan('0', '9');
{ Now assured of a digit so save the index, and look for
the closing comma }
isave:=1;
{ In case spaces between number and comma }
while (line[i-1] = ' ') do i:=i-1;

{ Fill in the discovered name }
lengthofname:=isave;
if (lengthofname > maxnamesize) then begin
lengthofname:=maxnamesize-1
end;
for k:=1 to lengthofname do begin
name.charvec[k]:=line[isave+k-1]
end;

{ Space-fill so as to allow string comparisons }
for k:=(lengthofname+1) to maxnamesize do begin
name.charvec[k):=''
end;
{ And fill in the rest of the record }
name.length:=lengthofname;
name.version:=digitized;

{ Now scan for the '=' sign that precedes the class }
scan('=','=');
{ And an alphabetic character following. The test may
also let through some non-alphabetics, but no matter. }
scan('A','Z');
{ Identify the class by its first letter. It is always
upper-case. }
if (line[i] = 'C') then class:=conformance
else if (line[i] = 'D') then class:=deviance
else if (line[i] = 'E') then class:=errorhandling
else if (line[i] = 'I') then class:=implementationdefined
else if (line[i] = 'Q') then class:=quality
else begin
{ Error, not recognized }
writeln(output);
writeln(output,'*****ERROR IN EXTRACT - WHAT CLASS?');
class:=other { default }
end;

{ Now we have established the desired values, so return }
end; { of procedure extract }

APPENDIX 1
PAGE A1-6

procedure scan(lowch,highch:char);
{
Scan moves the index i along the line until it finds
a character lying between lowch and highch inclusive.
It includes some simple error-handling which terminates
the program. }

var
state : loopcontrol;

begin { of scan }
{
Set loop to scan forwards }
state:=scanning;
{ Loop invariant R1 =
"characters from line[initial i] to line[i-1] are
not in the desired subrange." }
while (state = scanning) do begin
if (i > linelength) then begin
{ No more to go, so get out }
state:=notfound
end else if (line[i] >= lowch) and (line[i] <= highch)
then begin
state:=found
end else begin { Char is not in range }
i:=i+1
end; { of if }
end; { of while }

if (state = notfound) then begin
writeln(output);
writeln(output,'*****ERROR IN SCAN - REACHED LINE END');
goto 999 { in outer block and give up }
end;

{ Return, leaving i at found character }
end; { of procedure scan }
procedure newsuite;
{ This procedure may be used to initialize the run somehow, or to read in some parameters for the process. }
begin
  writeln(output, 'THIS IS AN EXECUTION OF THE SKELETON ',
          'VALIDATION SUITE PROCESSOR');
  writeln(output, '----------------------------------',
          '----------------------------------');
  page(output)
end; { of procedure newsuite }

procedure endsuite;
{ This procedure may be used to initiate a global job, or to check correct completion. }
begin
  writeln(output, 'SKELETON VALIDATION SUITE PROCESSOR END',
          ',count:4,',
          'TESTS READ')
end; { of procedure endsuite }

procedure newprogram(name : nametype;
    class: classtype;
    count: natural;
    var status: statustype);
{ This procedure is called at the recognition of a header comment while in dormant status. The first two parameters are derived from the header comment, while the third is simply the ordinal number of the test met in processing. The final parameter is the status of the search. }

The user's version has the responsibility of deciding what to do about this program by setting status to active or leaving it passive. In the first case all lines are processed by processline later, and the user may set up any headers, JCL statements, etc, beforehand. In the latter case the driver resumes searching for a header comment.

The name may be in digitized or alphabetized version - see procedure convert. Searching for a particular program can be done with a string comparison, for example:

`name.charvec = '6.2-1`

Searching for a subsection can be done with string comparisons if care is taken with collating sequence. On ASCII machines, space collates lower than anything else, so that

`(name.charvec >= '6.2') and
(name.charvec < '6.3')`

will determine all tests relevant to section 6.2 and its subsections.

The class may also be used in selection.

The count may be used to parcel up say 20 tests and run them in a batch, rather than the whole shebang at one go. An appropriate test is

`if ((count mod 20) = 0) then ...`

although this may be more appropriately used in endprogram. }

var
  i : linesize;
begin
  write(output, 'TEST PROGRAM '); 
  if (name.version = alphabetized) then convert(name);
  for i:=1 to name.length do write(output,name.charvec[i]);
  write(output, ' (ALIAS ') ;
  convert(name);
  for i:=1 to name.length do write(output,name.charvec[i]);
  writeln(output, ') , NO',count:4);
  writeln(output);
  writeln(output);
  status:=active { forcing print of all }
end; { of procedure newprogram }
procedure endprogram(name: nametype; class: classtype; count: natural); 

{ See the comments for newprogram. Endprogram can do exactly the same tests, but it has no responsibility for status which will automatically become dormant afterwards. }

begin 
  page(output)
  end; { of procedure endprogram }

procedure processline;
{ This procedure processes a source line of text, whatever that implies. Here we just print it. }

var 
  i : linesize;
begin 
  write(output,' ');
  for i:=1 to linelength do write(output,line[i]);
  writeln(output)
  end; { of procedure processline }

begin { of Main Program }
  count:=0;
  status:=dormant;
  newsuite; { Call in case user needs prologue }
  repeat begin until status = terminated 
    readline;
    if (status = dormant) then begin
      { We only look for header comments }
      if (linelength >= 2) then begin
        if (line[1] = 'I') and (line[2] = 'T') then begin
          extract;
          if (name.charvec = '999') then begin
            end
          end else begin
            count:=count+1;
          end
          { Newprogram may alter status too }
          newprogram(name,class,count,status)
        end
      end
      else begin
        processline;
      end
      if (linelength >= 4) then begin
        if (line[1] = 'e') and
          (line[2] = 'n') and
          (line[3] = 'd') and
          (line[4] = '.') then begin
          endprogram(name,class,count);
          status:=dormant
        end
      end
    end 
    end until (status = terminated);
endsuite; { Call in case user needs epilogue }

999:
end.
This program does not conform to the standard because its
meaning is altered by the truncation of its identifiers to 8
characters. Does the processor provide any indication that the
program does not conform?

Such surreptitious changes of meaning are dangerous.
Obviously processors with 8-character significance will have
difficulty in detecting such problems, but it can be done.
For processors with full significance it is easier.

program t5p2pdl(output);
onst
valueofaverylongidentifier1 = 10;
procedure p;
var
valueofaverylongidentifier2:integer;
begin
valueofaverylongidentifier2:11;
if valueofaverylongidentifier1 <>
valueofaverylongidentifier2 then
writeln(' IDENTIFIERS DISTINGUISHED...5.2.2-1')
else
writeln(' IDENTIFIERS NOT DISTINGUISHED...5.2.2-1')
ed;
endif;
end;

procedure x,procedure, procedure:char;
function x, function, function:integer;
iffy:boolean;
begin
procedure:='O';
procedure:='1';
procedure:=' 2 ,';
function:=o;
function:=l;
function :=2;
iffy:=true;
writeln(' PASS...6.1.2-3, IDENTIFIERS')
ed.

program t6p1p2d1(output);
var
i:(tick,cross,nil);
begin
i:=nil;
writeln(' DEVIATES...6.1.2-l, NIL')
ed.

program t6p1p2d2(output);
var
thing: (var,string);
begin
thing :=string;
writeln(' DEVIATES... 6.1.2-2, RESERVED
WORDS')
ed.

program t6p1p2d3(output);
var
procedurex,procedure,procedure:char;
functionx, function, function:integer;
iffy:boolean;
begin
procedure:='O';
procedure:='1';
procedure:=' 2 , ';
function:=o;
function:=l;
function :=2;
iffy:=true;
writeln(' DEVIATES...6.1.2-3, IDENTIFIERS')
ed.
[TEST 6.1.3-1, CLASS=CONFORMANCE]

{ The Pascal Standard permits identifiers to be of any length
  This test will simply print out 'PASS' if the compiler accepts
  identifiers of lengths up to 70 characters. }

program t6p3d1(output);
const
  begin
    writeln('FAIL...6.1.3-1');
  end.

program t6p3d2(output);
var
  begin
    writeln('PASS...6.1.3-2');
  end.

[TEST 6.1.3-2, CLASS=CONFORMANCE]

{ The Pascal Standard states that matching upper and lower
  case letters are equivalent in identifiers and word-symbols
  (i.e. reserved words) if they are permitted. If this is the
  case for this compiler, then the program shall print 'PASS'.
  This test is irrelevant for one case compilers. }

program t6p3d2(output);
var
  conform : integer;
begin
  writeln('PASS...6.1.3-2');
end.

[TEST 6.1.3-3, CLASS=QUALITY]

{ Although the Standard places no limit on the LENGTH of
  identifiers, they must be UNIQUE in at least the first 8
  characters. This test will determine the significance
  of identifiers from 3 characters to 20 characters.
  This program is of course non-standard, and relies on scope
  masquerades. It assumes a naive significance limit exists,
  or that there is none. Some compilers may violate the assumption
  by hashing an identifier tail or preserving the real length. }

program t6p3d3(output);
const
  begin
  end.

procedure signif;
begin
  end.

var
  x : integer;
begin
  end.
if \( x = 20 \) then
   writeln(' NUMBER OF SIGNIFICANT CHARACTERS >= 20')
else
   writeln(' NUMBER OF SIGNIFICANT CHARACTERS = ', x)
end;

begin
   signif;
end.

{TEST 6.1.5-1, CLASS=CONFORMANCE}

{ This program tests the conformance of the compiler to
  the syntax productions for numbers specified by the
  Pascal Standard.
  If all productions are permitted the program will
  print 'PASS'. The compiler fails if the program will
  not compile. }

program t6p1p5d1(output);
const
{ all cases are legal productions }
  a = 1;
  b = 12;
  c = 0123;
  d = 123.0123;
  e = 123.0123E+2;
  f = 123.0123E-2;
  g = 123.0123E2;
  h = 123E+2;
  i = 0123E-2;
  j = 0123E2;
begin
   writeln(' PASS...6.1.5-1')
end.

{TEST 6.1.5-2, CLASS=CONFORMANCE}

{ This program simply tests if very long numbers are permitted.
  The value should be representable despite its length. }

program t6p1p5d2(output);
const
  real = 123.456789012345678901234567890123456789;
begin
   writeln(' PASS...6.1.5-2')
end.

[TEST 6.1.5-3, CLASS=DEVIANCE]

{ The number productions specified in the Pascal Standard
  clearly state that a decimal point must be preceded by
  a digit sequence.
  The compiler deviates if the program compiles, in which case
  the program will print 'DEVIAATES', or if one of the cases is
  accepted.
  The compiler conforms if all the cases are rejected. }

program t6p1p5d3(output);
const
  r = .123;
var
  i : real;
begin
   i := .123;
   writeln(' DEVIATES...6.1.5-3')
end.

{TEST 6.1.5-4, CLASS=DEVIANCE}

{ The number productions specified in the Pascal Standard
  clearly state that a decimal point must be followed by
  a digit sequence.
  The compiler deviates if the program compiles, in which case
  the program will print 'DEVIATES'.
  The compiler conforms if the program fails to compile. }

program t6p1p5d4(output);
var
  i : real;
begin
   i := 0123.;
   writeln(' DEVIATES...6.1.5-4')
end.

{TEST 6.1.5-5, CLASS=DEVIANCE}

{ Spaces in numbers are forbidden by the Pascal Standard
  This includes spaces around '.' and 'E'. The compiler
  deviates if ONE or MORE of the cases below are accepted.
  The compiler conforms if ALL cases are rejected. }

program t6p1p5d5(output);
const
  one = 1 234;
  two = 0 .1234;
  three = 0 .1234;
  four = 123 4E2;
  five = 1234E 2;
  six = 1234E-2;
  seven = 1234E+ 2;
begin
   writeln(' DEVIATES...6.1.5-5')
end.
The Pascal standard allows equivalence of upper and lower-case letters in names and reserved words only. Will the compiler accept 'e' as equivalent to 'E'? It should not. The test is not relevant to one-case processors.

program t6plp5d6(output);
var
  i : real;
begin
  i :=123e2;
  writeln(' DEVIATES...6.1.5-6'}
end.

Labels are permitted in standard Pascal. This program simply tests if they are permitted by this compiler. The compiler fails if the program will not compile (or the message printed out is incorrect).

program t6plp6dl(output);
label
  1,2,3,4,5;
begin
  write(' P'}
  goto 4;
  1: write('.6'}
  goto 5;
  2: write('SS'}
  goto 3;
  3: write('...')
  goto 1;
  4: write('A'}
  goto 2;
  5: writeln('..1.6-1');
end.

Labels should be distinguished by their apparent integral value according to the Pascal Standard. This program tests if this is the case for this compiler. If so then the program shall print PASS.

program t6plp6d2(output);
label
  5,6,7;
begin
  goto 5;
  0006: goto 7;
  5: goto 6;
  0007: writeln('PASS...6.1.6-2')
end.

Character strings consisting of a single character are the constants of the standard type char. This program simply tests that these are permitted by the compiler. The compiler fails if the program will not compile.

program t6plp7dl(output);
const
  one = '1';
  two = '2';
var
  twotoo : char;
begin
  if (one <> twotoo) and (two = '2') then
  begin
    twotoo := '2';
    if twotoo = two then
      writeln('PASS...6.1.7-1')
    else
      writeln('FAIL...6.1.7-1')
  end
  else
    writeln('FAIL...6.1.7-1')
end.

The Pascal standard does not place an upper limit on the length of strings. This program tests if strings are permitted up to a length of 68 characters. The compiler fails if the program will not compile.

program t6plp7d2(output);
type
  string1 = packed array[1..68] of char;
  string2 = packed array[1..33] of char;
var
  alpha : string1;
  i : string2;
begin
  alpha := 'ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ
The Pascal standard allows quotes to appear as char constants and permits them to appear in strings. If this is desired, they must be written twice. This program tests that the compiler will allow this. The compiler fails if the program will not compile.

```pascal
program t6plp7d3(output);
const
quote = '';
strquote = 'CAN'T';
begin
  writeln(' PASS ... 6.1.7-3')
end.
```

This program tests that strings of different lengths are not compatible (i.e. 1..n and 1..n). The compiler fails if the program compiles.

```pascal
program t6plp7d4(output);
const
string1 = 'STRING1';
var
  string2 : packed array[1..5] of char;
begin
  string2 := string1;
  writeln(' DEVIATES ... 6.1.7-4')
end.
```

The Pascal Standard specifically states that character strings are constants of the type packed array[1..n] of char. This program tests that this type is not compatible with unpacked arrays. The compiler conforms if the program fails to compile.

```pascal
program t6plp7d5(output);
var
  string1 : packed array[1..4] of char;
  string2 : array[1..4] of char;
begin
  string1 := 'STR1';
  string2 := 'STR2';
  writeln(' DEVIATES ... 6.1.7-5')
end.
```

Again, as character strings are constants of the type packed array[1..n] of char, they should not be compatible with packed arrays of subranges of char. The compiler conforms if the program will not compile.

```pascal
program t6plp7d6(output);
var
  string1 : packed array[1..4] of char;
  string2 : packed array[0..3] of char;
  string3 : packed array[2..5] of char;
begin
  string1 := 'STR1';
  string2 := 'STR2';
  string3 := 'STR3';
  writeln(' DEVIATES ... 6.1.7-6')
end.
```

Again, as character strings are constants of the type packed array[1..n] of char, they should not be compatible with packed arrays of subranges of char. The compiler conforms if the program fails to compile.

```pascal
program t6plp7d7(output);
type
alpha = 'A'..'Z';
var
  string1 : packed array[1..4] of char;
  string2 : packed array[1..4] of alpha;
begin
  string1 := 'FOUR';
  string2 := 'FOUR';
  writeln(' DEVIATES ... 6.1.7-7')
end.
```
Similarly to 6.1.7-7, subranges of char should not be compatible with packed arrays of char. However, if the extension is allowed, it should be correctly handled.

Standards conforming processors will not compile the program, compilers which permit the extension should detect the error at compile-time or run-time, but should not execute without error.

Program t6plp7d8(output);
begin
end.

Program t6plp7d9(output);
var
  a = 'A';
begin
  a := string1;  [CASE 1]
end.

Program t6plp7d10(output);
var
  string1 : packed array[1..4] of char;
  string2 : packed array[1..4] of char;
begin
  writeln('DEVIATES...6.1.7-10');
end.

Program t6plp7d11(output);
begin
  writeln('DEVIATES...6.1.7-11');
end.

Program t6plp7d12(output);
begin
  writeln('DEVIATES...6.1.7-12');
end.

Program t6plp7d13(output);
begin
  writeln('DEVIATES...6.1.7-13');
end.
The Pascal Standard permits an open curly bracket to appear in a comment. This program tests that the compiler will allow this. The compiler fails if the program will not compile.

```pascal
program tpplp8d2(output);
begin
{ Is a { permitted in a comment?
writeln(' PASS...6.1.8-2')
end.
```

The compiler allows both forms of comments, must the delimiters be the same. If only one form of comment is permitted, the test is not relevant.

```pascal
program tpplp8d3(output);
begin
{ This is a standard comment }
(* This is an alternative form *)
{ What will happen here? *...*)
writeln(' PASS...6.1.8-3')
end.
```

In the case of an unclosed comment, does the compiler help the programmer to detect that this is so? Hard to trace run-time errors may occur if a comment accidentally encloses 1 or more statements.

```pascal
program tpplp8d4(output);
var
i : integer;
begin
i:=10;
{ Now write out the value of i.
writeln(' THE VALUE OF I IS!', i);
{ The value of i will not be printed because of the unclosed previous comment. }
i:=0
end.
```

Nested comments are not permitted in Pascal and hence this program should not compile. The compiler deviates if the program compiles and prints DEVIATES.

```pascal
program tpplp8d5(output);
begin
{ writeln(' RAN')
{ writeln(' RAN')
writeln(' DEVIATES...6.1.8-5, NESTED COMMENTS')
end.
```
This program checks that multiple repetitions are possible in the declaration part, and is provided as a check. Practically all occurrences will re-appear elsewhere in the validation suite.

program t6p2p1d2(output);
label
1,2,3;
const
one=1;
two=2;
three=3;
type
var
small = 1..3;
larger = 1..10;
biggest = 1..100;

var
tiny : small;
soso : larger;
big : biggest;

procedure p(var x small);
begin
x:=1
end;
procedure q(var y larger);
begin
y:=2
end;
procedure r(var z biggest);
begin
z:=3
end;
begin
p(tiny); goto 2;
r(big); goto 3;
r(soso); goto 1;
if (tiny=one) and (soso=two) and (big=three) then
   writeln(' PASS...6.2.1-2')
end.

{TEST 6.2.1-3, CLASS=DEVIANCE}

Checks to see that labels are not permitted unless they have been declared in the heading.

program t6p2p1d3(output);
begin
   writeln(' DEVIATES...6.2.1-3')
end.

{TEST 6.2.1-4, CLASS=DEVIANCE}

Checks to see that labels may not be given twice in the executable part. Since the label is not used in a goto this program is a stringent test.

program t6p2p1d4(output);
label
9;
begin
   9: writeln(' DEVIATES');
      if true <> false then
         9: writeln('...6.2.1-4')
   end.

{TEST 6.2.1-5, CLASS=DEVIANCE}

This program declares a label, but it is not sited nor referenced. This is illegal, as each declared label must appear once (and only once) in the executable part of the program.

program t6p2p1d5(output);
label
9;
begin
   writeln(' DEVIATES...6.2.1-5')
end.

{TEST 6.2.1-6, CLASS=CONFORMANCE}

This is the minimal program.

program t6p2p1d6;
begin
end.
The Pascal Standard states that '...local variables have values which are undefined at the beginning of the statement part. ...
The undefined value is dependent on the implementation. Ideally the program should not run. However, if it does, the program shall print the value of \( i \), whether it be a system initialized value or rubbish left over from procedure \( q \).

```pascal
program t5p2pld7(output);
procedure q;
var
  i, j : integer;
begin
  i := 2;
  j := 3
end;

procedure r;
var
  i : integer;
begin
  writeln('ERROR NOT DETECTED...6.2.1-7: THE VALUE OF I IS ', i)
end;

{ Program body }
begin
  q;
  r
end.
```

This test checks that a large number of types may be declared in a program. It is an attempt to discover any small limit imposed on the number of types by a compiler.

```pascal
program t6p2pld8(output);
type
t1 = 0 .. 1;
t2 = 0 .. 2;
t3 = 0 .. 3;
t4 = 0 .. 4;
t5 = 0 .. 5;
t6 = 0 .. 6;
t7 = 0 .. 7;
t8 = 0 .. 8;
t9 = 0 .. 9;
t10 = 0 .. 10;
t11 = 0 .. 11;
t12 = 0 .. 12;
t13 = 0 .. 13;
t14 = 0 .. 14;
t15 = 0 .. 15;
t16 = 0 .. 16;
t17 = 0 .. 17;
t18 = 0 .. 18;
t19 = 0 .. 19;
t20 = 0 .. 20;
t21 = 0 .. 21;
t22 = 0 .. 22;
t23 = 0 .. 23;
t24 = 0 .. 24;
t25 = 0 .. 25;
t26 = 0 .. 26;
t27 = 0 .. 27;
t28 = 0 .. 28;
t29 = 0 .. 29;
t30 = 0 .. 30;
t31 = 0 .. 31;
t32 = 0 .. 32;
t33 = 0 .. 33;
t34 = 0 .. 34;
t35 = 0 .. 35;
t36 = 0 .. 36;
t37 = 0 .. 37;
t38 = 0 .. 38;
t39 = 0 .. 39;
t40 = 0 .. 40;
t41 = 0 .. 41;
t42 = 0 .. 42;
t43 = 0 .. 43;
t44 = 0 .. 44;
t45 = 0 .. 45;
t46 = 0 .. 46;
t47 = 0 .. 47;
t48 = 0 .. 48;
t49 = 0 .. 49;
t50 = 0 .. 50;
var
  v1 : t1;
  v2 : t2;
```
{TEST 6.2.1-9, CLASS=QUALITY}

This test checks that a large number of labels may be declared
in a program. It is an attempt to detect a small compiler limit on
the number of labels.

program testpld9(output);
label
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
41, 42, 43, 44, 45, 46, 47, 48, 49, 50;
begin
1:
2:
3:
4:
5:
6:
7:
8:
9:
10:
11:
12:
13:
14:
15:
16:
17:
18:
19:
20:
21:
22:
23:
24:
25:
26:
27:
28:
29:
30:
31:
32:
33:
34:
35:
36:
37:
38:
39:
40:
41:
42:
43:
44:
45:
46:
47:
48:
49:
50:
begin
write(' 50 TYPES COMPILED...6.2.1-8')
end.
The Pascal Standard permits redefinition of a user name, by a further defining occurrence in a range (eg. procedure block) enclosed by the first defining occurrence. This second range (and all ranges enclosed by it) are excluded from the scope of the defining occurrence of the first range. This program tests the scope conformance of the compiler for user names.}

program tip2p2d1(output);

const
range = 10;

var
i : integer;
pass : boolean;
procedure redefine;

begin
i := range;
end;

begin
true := false;
if true = false then
writeh(‘FAIL...6.2.2-1’)
else
writeh(‘PASS...6.2.2-1’)
end.

{TEST 6.2.2-1, CLASS=CONFORMANCE}

[ The Pascal Standard allows a user to redefine a predefined name. This program tests whether this is allowed by this compiler. ]

program tip2p2d2(output);

var
true : boolean;
begin
true := false;
if true = false then
writeh(‘FAIL...6.2.2-2’)
else
writeh(‘PASS...6.2.2-2’)
end.

{TEST 6.2.2-2, CLASS=CONFORMANCE}

The Pascal Standard permits redefinition of a user name, by a further defining occurrence in a range (eg. procedure block) enclosed by the first defining occurrence. This second range (and all ranges enclosed by it) are excluded from the scope of the defining occurrence of the first range. This program tests the scope conformance of the compiler for user names.}

program tip2p2d3(output);

type
node = real;

procedure ouch;

begin
new(ptr);
ptr := true;
writeh(‘PASS...6.2.2-3’)
end.

begin
ouch;
end.

{TEST 6.2.2-3, CLASS=CONFORMANCE}

[ This program is similar to 6.2.2-4, however a type identifier, say T, which specifies the domain of a pointer type TT, is permitted to have its defining occurrence anywhere in the type definition part in which TT occurs. Thus in this example, (node=real)'s scope is excluded from the type definition of ouch. The compiler fails if the program does not compile or fails at run time. ]

program tip2p2d4(output);

type
node = real;

procedure ouch;

begin
new(ptr);
ptr := true;
writeh(‘PASS...6.2.2-4’)
end.

begin
ouch;
end.

{TEST 6.2.2-4, CLASS=CONFORMANCE}

[ The Pascal Standard permits redefinition of a user name, by a further defining occurrence in a range (eg. procedure block) enclosed by the first defining occurrence. This second range (and all ranges enclosed by it) are excluded from the scope of the defining occurrence of the first range. This program tests the scope conformance of the compiler for user names. ]
[The Pascal Standard says: that the defining occurrence of an identifier or label precedes all corresponding occurrences of that identifier or label in the program text (except for specific pointercase). The scope of an identifier or label also includes the whole block in which it is defined, thereby disallowing any references to an outer identifier of the same name preceding the defining occurrence. Some compilers may not conform to this and allow some scope overlap. The compiler conforms if the program does not compile and objects to the use of 'red' in ouch preceding its definition. ]

program t6p2p2d4(output);
const
red = 1;
violet = 2;
procedure ouch;
const
m = red;
n = violet;
type
a = array[m..n] of integer;
var
v : a;
colour : (yellow,green,blue,red,indigo,violet);
begin
v[l]:=1;
colour:=red;
ouch;
writeln('DEVIATES...6.2.2-4 -> SCOPE ERROR NOT DETECTED')
end.

[ Similarly to 6.2.2-2, labels are allowed to be redefined in a range enclosed by the first defining occurrence (eg. procedures and functions). This program tests if this is permitted by this compiler. ]

program t6p2p2d5(output);
label
4,5,6;
var
j : integer;
procedure redefine;
label
6,7,8;
var
j : integer;
begin
j:=l;
goto 6;
7: j:=j-1;
goto 8;
6: j:=j+1;
goto 7;
8: j:=0;
end;
begin
goto 4;
5: i:=i+l;
goto 6;
4: i:=l;
redefine;
goto 5;
6: if i=l then
writeln('FAIL...6.2.2-5')
else
writeln('PASS...6.2.2-5')
end.

PASCAL NEWS #16
OCTOBER, 1979
PAGE 36
As for the other conformance tests in this section, it is possible to redefine a field-name of a record within the same scope as this record. The compiler also fails if the program does not compile.

```pascal
program t6p2p2d5(output);
var
  j: integer;
  x: record
    j: integer
  end;

begin
  j:=1;
  x.j:=2;
  with x do
      j:=3;
  if (j=1) and (x.j=3) then writeln('PASS ••• 6.2.2-6')
  else writeln('FAIL ••• 6.2.2-6')
end.
```

It is possible to redefine a function name within the scope of a function name. This test checks that the inner function redefines f, whether an erroneous assignment to f is detected or whether the erroneous outer f, with no function assignment is allowed to execute.

```pascal
program t6p2p2d8(output);
var
  j, k: integer;
  function fl(i:integer): integer;
  function f2(i:integer): integer;
  function f3(i:integer): integer;

begin
  f3:=1;
  fl:=i
end;

begin
  f2:=f3(i)
end;

begin
  j:=f2(1)
end;

begin
  k:=fl(5);
  if (k=5) then writeln('PASS ••• 6.2.2-8, FUNCTION')
  else writeln('FAIL ••• 6.2.2-8, FUNCTION')
end.
```
program tnp2p2d9(output);
var
  i : integer;
function fl : integer;
begin
  fl := 6
end;
function f2(i : integer) : integer;
begin
  f2 := i;
  fl := 5
end;
begin
  i := fl;
i := f2(2);
writeln('   DEVIATES...6.2.2-9, FUNCTION')
end.

program tnp2p2d10(output);
type
  colour = (red, amber, green);
var
  c : colour;
procedure nested;
type
  colour = (purple, red, blue);
var
  paint : colour;
begin
  c := green;
paint := red;
c := pred(amber);
if (ord(c) <> 0) or (ord(paint) <> 1) then
  writeln(' FAIL...6.2.2-10, SCOPE');
end;
begin
  nested;
  if (c <> red) then
    writeln(' FAIL...6.2.2-10, SCOPE')
  else
    writeln(' PASS...6.2.2-10, SCOPE')
end.

program tnp3d1(output);
const
  ten = 10;
minusten = -10;
minustentoo = -ten;
de cade = ten;
dot = '.';
st ars = '**';
on = true;
pl = 3.1415926;
minuspi = -pl;
begin
  writeln(' PASS...5.3-1')
end.
[TEST 6.3-2, CLASS=DEVIANCE]

This program checks that signed chars are not permitted. Note that minus may have a worse effect than plus.

```pascal
program t6p3d2(output);
const
dot = '.';
plustdot = + dot;
begin
  writeln(' DEVIATES...6.3-2')
end.
```

[TEST 6.3-3, CLASS=DEVIANCE]

This program checks that signed strings are not permitted. Note that minus may have a worse effect than plus.

```pascal
program t6p3d3(output);
const
stars = '****';
plusstars = + stars;
begin
  writeln(' DEVIATES...6.3-3')
end.
```

[TEST 6.3-4, CLASS=DEVIANCE]

This program checks that signed scalars are not permitted. Note that minus may have a worse effect than plus.

```pascal
program t6p3d4(output);
const
truth = true;
plustruth = + truth;
begin
  writeln(' DEVIATES...6.3-4')
end.
```

[TEST 6.3-5, CLASS=DEVIANCE]

This program tests that signed constants are not permitted in other contexts than const declarations.

```pascal
program t6p3d5(output);
const
dot = '.';
begin
  writeln(' DEVIATES', +dot, '...6.3-5')
end.
```

[TEST 5.3-6, CLASS=DEVIANCE]

A constant may not be used in its own declaration - the following is a pathological case which should be detected or at least handled with care.

```pascal
program t6p3d6(output);
const
ten = 10;
procedure p;
const
ten = ten;
begin
  if ten=10 then
    writeln(' DEVIATES...6.3-6: SCOPE ERROR')
  else
    writeln(' DEVIATES...6.3-6: DEFINITION POINT ERROR')
end;
begin
  p
end.
```

[TEST 6.4.1-1, CLASS=CONFORMANCE]

This program tests to see that plinter types can be declared anywhere in the type part. This freedom is explicitly permitted in the standard.

```pascal
program t6p4d1d1(output);
type
  ptr1 = T pllar;
  pllar = record r,theta
    plurelink = T purelink;
  end;
  ptr2 = T person;
  ptr3 = ptr2;
  person record
    mother,father : ptr2;
    firstchild : ptr2;
    nextsibling : ptr3
  end;
begin
  mother,father
  firstchild
  nextsibling
  writeln(' PASS...6.4.1-1')
end.
```
This program tests that attempts to use types in their own definitions are detected. Two examples are attempted. Both should fail.

program t6p4p4ld2(output);
  type
    x = record
      xx : x
    end;
    y = array[0..1] of y;
  begin
    writeln(' DEVIATES...6.4.1-2')
  end.

This program also tests that attempts to use types in their own definitions are detected, but inserts a nasty scope twist by making another type with the same identifier available in an outer scope. It should be excluded from this scope, according to the Standard.

program t6p4p4ld3(output);
  type
    x = integer;
  procedure p;
    type
      x = record
        y : x
      end;
    begin
      writeln(' DEVIATES...6.4.1-3: SCOPE ERROR')
    end;
  begin
    p
  end.

This program tests that the standard simple types have all been implemented. They are denoted by predefined type identifiers. The compiler fails if the program does not compile.

program t6p4p4p2p2d2(output);
  var
    a, b, c, d : integer;
    real;
    boolean;
    char;
  begin
    a := 6*2 + 3;
    b := 3.14159*2;
    c := (a = 15);
    d := 'Z';
    writeln(' PASS...6.4.2.2-1')
  end.

The Pascal Standard specifies that the values an integer may take are within the range -maxint..+maxint. This program checks this.

program t6p4p4p2p2d3(output);
  type
    natural = 0..maxint;
    whole = -maxint..+maxint;
  var
    i : natural;
    j : whole;
    k : integer;
  begin
    i := maxint;
    j := maxint;
    k := maxint;
    writeln(' PASS...6.4.2.2-2')
  end.

The Pascal Standard states that type BOOLEAN has truth values denoted by the identifiers true and false, and that they are such that false is less than true. This program tests if the compiler allows this.

program t6p4p4p2p2d4(output);
  if false < true then
    writeln(' PASS...6.4.2.2-3')
  else
    writeln(' FAIL...6.4.2.2-3')
  end.
program tlp4p2p2d4(output);
var
begin
begin
end.

begin
var a,b:boolean;
begin
if a then
else
writeln('FAI...6.4.2.2-6')
else
writeln('PASS...6.4.2.2-6')
end.

[TEST 6.4.2.2-5, CLASS=CONFORMANCE]
{ The Pascal Standard states that the upper-case letters A-Z are
ordered, but not necessarily contiguous.
This program determines if this is so, and prints
a message as to whether the compiler passes or not. }

program tlp4p2p2d5(output);
begin
if ("A" < "B") and ("B" < "C") and ("C" < "D") and
("D" < "E") and ("E" < "F") and ("F" < "G") and
("G" < "H") and ("H" < "I") and ("I" < "J") and
("J" < "K") and ("K" < "L") and ("L" < "M") and
("M" < "N") and ("N" < "O") and ("O" < "P") and
("P" < "Q") and ("Q" < "R") and ("R" < "S") and
("S" < "T") and ("T" < "U") and ("U" < "V") and
("V" < "W") and ("W" < "X") and ("X" < "Y") and
("Y" < "Z") then
writeln('PASS...6.4.2.2-5')
else
writeln('FAIL...6.4.2.2-5: NO ORDERING')
end.

[TEST 6.4.2.2-6, CLASS=CONFORMANCE]
{ The Pascal Standard states that the lower-case letters a-z are
ordered, but not necessarily contiguous.
This program determines if this is so, and prints
a message as to whether the compiler passes or not. }

NOTE: this program uses lower-case char constants and may
fail for this reason. The test is also irrelevant for
one-case compilers. }

program tlp4p2p2d6(output);
begin
if ('a' < 'b' and ('b' < 'c') and ('c' < 'd') and
('d' < 'e') and ('e' < 'f') and ('f' < 'g') and
('g' < 'h') and ('h' < 'i') and ('i' < 'j') and
('j' < 'k') and ('k' < 'l') and ('l' < 'm') and
('m' < 'n') and ('n' < 'o') and ('o' < 'p') and
('p' < 'q') and ('q' < 'r') and ('r' < 's') and
('s' < 't') and ('t' < 'u') and ('u' < 'v') and
('v' < 'w') and ('w' < 'x') and ('x' < 'y') and
('y' < 'z') then
writeln('PASS...6.4.2.2-6')
else
writeln('FAIL...6.4.2.2-6: NO ORDERING')
end.

[TEST 6.4.2.2-7, CLASS=IMPLEMENTATIONDEFINED]
{ The Pascal Standard states that the value of maxint is
dependent on the implementation.
This program prints out the implementation defined value
of maxint. }

program tlp4p2p2d7(output);
begin
writeln('THE IMPLEMENTATION DEFINED VALUE OF MAXINT IS
maxint')
end.

[TEST 6.4.2.3-1, CLASS=CONFORMANCE]
{ This program checks the possible syntax productions for
enumerated types, as specified by the Pascal Standard.
The compiler fails if the program does not compile. }

program tlp4p2p3d1(output);
type
  singularitytype = (me);
  switch = (on/off);
  maritalstatus = (married/divorced/widowed/single);
  colour = (red/pink/orange/yellow/green);
  cardsuit = (heart/diamond/spade/club);

var
i: singularitytype;
begin
  i:=me;
  writeln('PASS...6.4.2.3-1')
end.

[TEST 6.4.2.3-2, CLASS=IMPLEMENTATIONDEFINED]
{ The Pascal Standard states that the values of a range of
digits 0..9 are ordered and contiguous.
The program tests these two criteria for these characters. }

program tlp4p2p2d8(output);
begin
if ('0' < '1') and
('1' < '2') and
('2' < '3') and
('3' < '4') and
('4' < '5') and
('5' < '6') and
('6' < '7') and
('7' < '8') and
('8' < '9')

{TEST 6.4.2.2-4, CLASS=CONFORMANCE}
{ The Pascal Standard states that the character values representing
the digits 0..9 are ordered and contiguous.
The program tests these two criteria for these characters. }
program tlp4p2p2d4(output);
var
begin
begin
end.

if ('A' < 'B') and ('B' < 'C') and ('C' < 'D') and
('D' < 'E') and ('E' < 'F') and ('F' < 'G') and
('G' < 'H') and ('H' < 'I') and ('I' < 'J') and
('J' < 'K') and ('K' < 'L') and ('L' < 'M') and
('M' < 'N') and ('N' < 'O') and ('O' < 'P') and
('P' < 'Q') and ('Q' < 'R') and ('R' < 'S') and
('S' < 'T') and ('T' < 'U') and ('U' < 'V') and
('V' < 'W') and ('W' < 'X') and ('X' < 'Y') and
('Y' < 'Z') then
writeln('PASS...6.4.2.2-5')
else
writeln('FAIL...6.4.2.2-5: NO ORDERING')
end.

program tlp4p2p2d5(output);
begin
if ('A' < 'B') and ('B' < 'C') and ('C' < 'D') and
('D' < 'E') and ('E' < 'F') and ('F' < 'G') and
('G' < 'H') and ('H' < 'I') and ('I' < 'J') and
('J' < 'K') and ('K' < 'L') and ('L' < 'M') and
('M' < 'N') and ('N' < 'O') and ('O' < 'P') and
('P' < 'Q') and ('Q' < 'R') and ('R' < 'S') and
('S' < 'T') and ('T' < 'U') and ('U' < 'V') and
('V' < 'W') and ('W' < 'X') and ('X' < 'Y') and
('Y' < 'Z') then
writeln('PASS...6.4.2.2-5')
else
writeln('FAIL...6.4.2.2-5: NO ORDERING')
end.

program tlp4p2p2d6(output);
begin
if ('a' < 'b' and ('b' < 'c') and ('c' < 'd') and
('d' < 'e') and ('e' < 'f') and ('f' < 'g') and
('g' < 'h') and ('h' < 'i') and ('i' < 'j') and
('j' < 'k') and ('k' < 'l') and ('l' < 'm') and
('m' < 'n') and ('n' < 'o') and ('o' < 'p') and
('p' < 'q') and ('q' < 'r') and ('r' < 's') and
('s' < 't') and ('t' < 'u') and ('u' < 'v') and
('v' < 'w') and ('w' < 'x') and ('x' < 'y') and
('y' < 'z') then
writeln('PASS...6.4.2.2-6')
else
writeln('FAIL...6.4.2.2-6: NO ORDERING')
end.
The Pascal Standard states that the ordering of the values of the enumerated type is determined by the sequence in which the constants are listed, the first being before the last. The compiler fails if the program does not compile.

```pascal
program t6p4p2p3d2(output);
var
  suit : (club, spade, diamond, heart);
  a : boolean;
  b : boolean;
begin
  a := (succ(club) = spade) and
       (succ(spade) = diamond) and
       (succ(diamond) = heart);
  b := (club < spade) and
       (spade < diamond) and
       (diamond < heart);
  if a and b then
    writeln('PASS...6.4.2.3-2')
  else
    writeln('FAIL...6.4.2.3-2')
end.
```

This program tests that a type may be defined as a subrange of another ordinal-type (host-type). The compiler fails if one or more of the cases below are rejected.

```pascal
program t6p4p2p4d1(output);
var
  colour = (red, pink, orange, yellow, green, blue);
  somecolour = red..green;
  century = 1..100;
  twotynse = -10..+10;
  digits = '0'..'9';
  zero = 0.0;
  logical = false..true;
begin
  tf : logical;
begin
  tf := true;
  writeln('PASS...6.4.2.4-1')
end.
```

This program tests to see if real constants are permitted in a subrange declaration. The Pascal Standard states that a subrange definition must be of a subrange of another ordinal type. This rules out real constants in the definition.

```pascal
program t6p4p2p4d2(output);
type
  wiregauge = 0.001..0.2;
begin
  writeln('DEVIATES...6.4.2.4-2')
end.
```

This program tests the compilers' conformance to this point. The compiler conforms if both the cases are rejected.

```pascal
program t6p4p2p4d3(output);
type
  mixedup = 100..0;
  reverse = 'Z'..'A';
begin
  writeln('DEVIATES...6.4.2.4-3: EMPTY SUBRANGES ALLOWED')
end.
```

The Pascal Standard states that only structured types may be PACKED (array, set, file and record types). This program tests this point. The compiler conforms if the program will not compile.

```pascal
program t6p4p3p1d1(output);
type
  switch = packed(on, off);
  state = packed(high, low, invalid);
  decade = packed 0..10;
begin
  writeln('DEVIATES...6.4.3.1-1: IMPROPER USE OF PACKED')
end.
```
The Pascal Standard states that a structured type identifier may not be used in a PACKED type definition. The compiler passes if the program fails to compile.

```pascal
program t6p4p3p1d2(output);
type
complex = record
  realpart : real;
  imagpart : real;
end;
packcom = packed complex;

begin
  writeln('DEVIATES...6.4.3.1-2 : IMPROPER USE OF PACKED')
end.
```

The Pascal Standard allows array, set, file and record types to be declared as PACKED. The program simply tests that all these are permitted. The compiler fails if the program will not compile.

```pascal
program t6p4p3p1d3(output);
type
  urray = packed array[1..10] of char;
  rekord = packed record
    bookcode : integer;
    authorcode : integer;
  end;
  fyle = packed file of urray;
  card = (heart,diamond,spade,club);
  sett = packed set of card;

begin
  writeln('PASS...6.4.3.1-3')
end.
```

This program tests all the valid productions for an array declaration from the syntax specified by the Pascal Standard. The compiler fails if one or more cases are rejected.

```pascal
program t6p4p3p2dl(output);
type
cards = {'two','three','four','five','six','seven','eight','nine','ten','jack','queen','king','ace'};
suit = {'heart','diamond','spade','club'};
hand = array[cards] of suit;
picturecards = array[jack,king] of suit;
played = array[cards] of array[heart,diamond] of boolean;
playedtoo = array[cards,heart,diamond] of boolean;
begin
  writeln('PASS...6.4.3.2-1')
end.
```
As mentioned in 6.4.3.2-3, an index type is an ordinal type, thus \texttt{INTEGER} may appear as an index type. However on most machines this would represent an unusually large array, and thus may not be allowed by the compiler.

This program tests if such a declaration is permitted, and if not, is the diagnostic appropriate.

\begin{verbatim}
program t6p4p3p2d4(output);
type
  everything = array[integer] of integer;
var
  all : everything;
begin
  all[maxint] := 1;
  all[0] := 1;
  all[-maxint] := 1;
  writeln('QUALITY...6.4.3.2-4: -->INTEGER BOUNDS PERMITTED')
end.
\end{verbatim}

Strings must have a subrange of integers as an index type. The compiler deviates if this program compiles and prints \texttt{DEVIATES}.

\begin{verbatim}
program t6p4p3p2d5(output);
type
  colour = (red,blue,yellow,green);
  c11 = blue..green;
var
  s : packed array[c11] of char;
begin
  s := 'ABC';
  writeln('DEVIATES...6.4.3.2-5, INDEX TYPE')
end.
\end{verbatim}

This program simply tests that all valid productions from the syntax for record types (as specified by the Pascal Standard) are accepted by this compiler. The compiler fails if one or more cases are rejected.

\begin{verbatim}
program t6p4p3p3d1(output);
type
  string = packed array[1..25] of char;
  married = (false,true);
  shape = (triangle,rectangle,square,circle);
  angle = 0..90;
  a = record
    year : integer;
    month : 1..12;
    day : 1..31;
  end;
  b = record
    name,firstname : string;
    age : 0..99;
    case married of
      true: (spousename : string);
      false : ()
    end;
  c = record
    case s : shape of
      triangle : (side : real;
                  inclination,angle1,angle2 : angle);
      square,rectangle : (side1,side2 : real;
                           skew,angle3 : angle);
      circle : (diameter : real)
    end;
  end;
begin
  writeln('PASS...6.4.3.3-1')
end.
\end{verbatim}
The Pascal Standard states that the occurrence of a field identifier within the identifier list of a record section is its defining occurrence as a field identifier for the record type in which the record section occurs. This should allow redefinition of a field identifier in another type declaration. The compiler fails if the program does not compile.

```pascal
program t6p4p3p3d2(output);
type
  a = record
    realpart: real;
    imagpart: real;
  end;
rea1part = (notimaginary,withbody,withsubstance);
begin
  writeln(' PASS ••• 6.4.3.3-2')
end.
```

The Pascal Standard permits the declaration of an empty record, this empty record serves little purpose, and for this reason some compilers will not allow it to be used. The compiler fails if the program does not compile.

```pascal
program t6p4p3p3d3(output);
type
  statuskind = (defined,undefined);
  anptykind = record end;
var
  empty : anptyklnd;
  ntl1lber:
begin
  case status:statuskind of
    defined (i integer) ;
    undefined: (e : emptykind)
  end;
  writeln(' PASS ••• 6.4.3.3-4')
end.
```

Similarly to 6.4.3.3-2, a tag-field may be redefined elsewhere in the declaration part. The compiler fails if the program will not compile.

```pascal
program t6p4p3p3d4(output);
type
  var
    which = (white,black,warlock,sand);
  po1ex : record
    case which:bool~an of
      true: (realrk'lrt:real;
        imagpart:rea1i;
      false: (theta,~ea1;
        magni,
        real)
    end;
begin
  po1ex.which:=true;
  po1ex .realpart:=O.5,
  po1ex. imagpart:=O.8;
  writeln(' PASS ••• 6.4.3.3-4')
end.
```

The Pascal Standard states that if a change of variant occurs (by assigning a value associated with a variant to the tag-field), then the fields associated with the previous variants cease to exist. This program causes the error to occur.

```pascal
program t6p4p3p3d5(output);
type
  var
    two = (a,b);
  variant : record
    case tagfield:two of
      a: (m:integer);
      b: (n: integer)
    end;
end;
```

```
begin
  variant.tagfield:=a;
  variant.m:=l;
  i:=variant.n; {illegal}
  writeln(' ERROR NOT DETECTED ••• 6.4.3.3-5')
end.
```
The program causes an error by accessing a field with an undefined value. The undefined arises because when a change of variant occurs, those fields associated with the new variant come into existence with undefined values.

```pascal
program t6p4p3p3d6(output);
type
tYo'O = (a ,b) ;
var
variant : record
case tagfield:two of
  a (m integer;  
  l integer);
  b (n integer;  
  o integer)
end;
i integer;
begin
variant.tagfield:=a;
variant.m:=l;
variant.l:=l;
variant.tagfield:=b;
variant.n:=l;
n:=variant.o; { illegal}
writeln(' ERROR NOT DETECTED...6.4.3.3-6')
end.
```

This test is similar to 6.4.3.3-5, except that no tag-field is used. A change of variant occurs by reference to a field associated with a new variant. Again, these fields come into existence with undefined values. The compiler conforms if the program does not compile.

```pascal
program t6p4p3p3d7(output);
type
tYo'O = (a,b);
var
variant : record
case two of
  a (m:integer;  
  l:integer);
  b (n:integer;  
  o:integer)
end;
i : integer;
begin
variant.n :=1;
variant.o :=1;
invariant.m:=l;
i:=variant.o; [ illegal]
writeln(' ERROR NOT DETECTED...6.4.3.3-8')
end.
```

This test is similar to 6.4.3.3-5, except that no tag-field is used. Variant changes occur implicitly as a result of assignment to fields. The fields associated with the new variant come into existence with undefined values.

```pascal
program t6p4p3p3d7(output);
type
two = (a,b);
var
variant : record
case two of
  a (m:integer); 
  b (n : integer);
end;
i : integer;
begin
variant.n :=2;
i:=variant.n; [ illegal]
writeln(' ERROR NOT DETECTED...6.4.3.3-7')
end.
```
Note this program relies on the compiler deviating for tests 6.4.3.3-5 to 6.4.3.3-8. If the compiler conforms for these tests, this program will not compile/run.

The method of storage for fields of variants may differ, depending on the method of definition. Programmers should not rely on the values of fields under one variant still being accessible from another. However, the relationships between the two variants in this example may be determined by the output of the program.

```pascal
program t6p4p3p3d9(output);
type
two = (a,b);
variant : record
case tagfield : two of
  a: (i,j,k : integer);
  b: (l : integer;
    m : integer;
    n : integer)
end;
begin
  variant.tagfield:=a;
  variant.i:=I;
  variant.j:=2;
  variant.k :=3;
  variant.tagfield:=b;
  if (variant.I=1) and (variant.m=2) then
    writeln(' EXACT CORRELATION- I:L J:M K:N')
  else
    if (variant.I=3) and (variant.m=2) then
      writeln(' REVERSE CORRELATION - I:N J:M K:L')
    else
      writeln(' UNKNOWN CORRELATION - I:m are:' ,
        variant.I,variant.m,variant.n)
end.
```

The Pascal Standard states that case constants must be distinct, and are of an ordinal type which is compatible with the tag-field. This program tests to see if the compiler will permit case constants outside the tag-field subrange - it should.

The compiler passes if the program runs. A warning might be appropriate, however, as fields outside the tag-field subrange are not accessible.

```pascal
program t6p4p3p3d10(output);
type
  a = 0..3;
  b = record
    case c:a of
      0: (d:array[1..21] of boolean);
      1: (e:array[1..31] of boolean);
      2: (f:array[1..4] of boolean);
      3: (g:array[1..5] of boolean);
      4: (h:array[1..I] of boolean)
    end;
begin
  case c:a of
    0: (d:array[1..21] of boolean);
    1: (e:array[1..31] of boolean);
    2: (f:array[1..4] of boolean);
    3: (g:array[1..5] of boolean);
  end;
  writeln(' PASS ...6.4.3.3-10')
end.
```

This program is similar to 6.4.3.3-3, except an empty record is assigned a value. The program conforms if the program does not compile or run.

```pascal
program t6p4p3p3d11(output);
type
  statuskind = (defined,undefined);
  emptykind = record
    end;
var
  empty : emptykind;
  number : record
    var
      status:statuskind of
        defined : (i : integer);
        undefined: (e : emptykind)
    end;
begin
  with number do
  begin
    status:=undefined;
    writeln(' PASS ...6.4.3.3-11')
  end.
end.
```
program t6p4p3p4d12(output);
var
type
statuskind = (defined,undefined);
emptykind = record empty
    number: record
        case status:statuskind of
            defined : (i : integer);
            undefined: (e : emptykind)
        end;
    end;
end.

begin
    with number do
        begin
            status:=undefined;
            e:=empty { undefined despite being empty }
        end;
    writeln(' IMPLEMENTATION DOES NOT ALLOW SET OF CHAR')
end.

program t6p4p3p3d13(output);
type
    a=record
        case b:boolean of
            true: (c:char);
            false: (case c:char of
                true: (e: emptykind))
        end;
    var
        g:s;
    begin
        g:=false;
        g.d:=false;
        g.f:=false;
        writeln(' PASS...6.4.3.3-13, VARIANTS')
    end.

begin
program t5p4p3p3d12(output);
begin
end.

{ TEST 6.4.3.3-12, CLASS=ERRORHANDLING }
{ This program is similar to 6.4.3.3-3, except here an error is caused by assigning the undefined value of the variable empty to the field e. This error should be detected. }

{ TEST 6.4.3.4-1, CLASS=CONFORMANCE }
{ This program simply tests that set types as described in the Pascal Standard are permitted. The compiler fails if the program will not compile. }
{ TEST 6.4.3.4-2, CLASS=IMPLEMENTATIONDEFINED }
{ This program tests if a set of char is permitted by the compiler. }
{ TEST 6.4.3.4-3, CLASS=IMPLEMENTATIONDEVIANCE }
{ The Pascal Standard states that the base-type of the range of a set must be an ordinal-type. This should eliminate sets with real and structured ranges. Some compilers may allow these and hence will deviate for those cases not flagged as errors. }
The Pascal Standard states that the largest and smallest values permitted in the base-type of a set-type are implementation defined. The size of the base-type permitted may be determined by examining which of the cases below are accepted by the compiler.

```
program t6p4p3p4d4(output);
type
  setone = set of -1..1;
  settwo = set of char;
  setthree = set of 0..1000;
  setfour = set of 0..10;
  setfive = set of 0..30;
  setsix = set of 0..30;
  sevenc = set of 0..40;
  seteight = set of 0..50;
  setnine = set of 0..60;
  setten = set of 0..70;
var
  s: setthree;
begin
  s:=[1000];
  writeln(' IMPLEMENTATIONDEFINED...6.4.3.4-4 -->','
          GOOD IMPLEMENTATION OF SETS');
end.
```

This test is an implementation of Warshall's algorithm in Pascal. It serves to give a program which can be used both to time a simple procedure using sets, and which can measure the space requirements. In both cases the measurements of the procedure Warshall are to be compared.

By way of comparison, the Tasmanian compiler on the Burroughs B6700 yielded:
- space = 143 bytes (6864 bits)
- time = 0.016461 seconds

```
program t6p4p3p4d5(output);
const
  size = 79; {array is (size+1) by (size+1) sq\!ard
            r=4;{size div 16:
            bitsperword::=16;
            bitsminusl = 15; {bits per word allow,3 this}
  type
    btype = array [0..size] of set of 0..bitsminus1;
var
  seed: integer;
  t1,t2:real;
  original,closure:btype;
function generate:integer;
begin
  seed:=57*seed+1;
  generate := (seed mod
               .::::--1));
  seed:=seed mod 571
end; {of generate}

procedure fill(var a:btype;p:integer);
var
  i,j:0..size;
begin
  for j:=0 to p do begin
    a[0][j]:=[];
    for i:=1 to size do a[i]:=a[0];
    for k:=1 to p do begin
      i:=generate;
      l:=generate;
      j:=i div bitsperword;
      a[i][j] := a[i][j]+(l mod bitsperword)
    end;
  end;
end; {of fill}

procedure print(var b:btype);
var
  i,j:0..size;
begin
  for i:=0 to size do begin
    write(' ',
          if (j mod bitsperword) in b[i][j div bitsperword] then
      write('+')
      else
      write('-')
    end;
end.
```
procedure warshallsalgorithm(var a,b:btype);
end.

begin
  writeln('TIME=' ,t2-t1);
  warshallsalgorithm(original,closure);
  writeln(original);
  writeln(closure);
  if i mod bitsperword then
    for j=0 to size do
      if (j mod bitsperword) in b[j] then
        for k=0 to words do
          b[j][k] := b[j][k] + b[i][k];
    end;
end;  

begin
  if i=0 to size do
    if j=0 to size do
      if (i mod bitsperword) in b[j] then
        for k=0 to words do
          b[j][k] := b[j][k] + b[i][k];
    end;
  end;

begin
  writeln(' ORIGINAL MATRIX');
  writeln(' TRANSITIVE CLOSURE');
  end;

begin
  fiel : file of char;
  file2 : file of real;
  file3 : file of record
    a : integer;
    b : boolean
  end;

begin
  writeln(' PASS...6.4.3.5-1')
end.

begin
  readline;
  writeln(file,'ABC');
  reset(file);
  if eof(file) then get(file);
else
  writeln(file,'FAIL...6.4.3.5-2')
else
  writeln(file,'PASS...6.4.3.5-2')
end.

begin
  rewrite(file);
  writeln(file,'FAIL...6.4.3.5-3')
else
  writeln(file,'PASS...6.4.3.5-3')
end.

begin
  fiel : file of char;
  file2 : file of real;
  file3 : file of record
    a : integer;
    b : boolean
  end;

begin
  writeln(file,'FAIL...6.4.3.5-3')
else
  writeln(file,'PASS...6.4.3.5-3')
end.
This program tests if an end-of-line marker is inserted at the end of the line on the predefined file output, if not explicitly done in the program (i.e. is the buffer flushed). See also test 6.4.3.5-3.

```pascal
program t6p4p3p5d4(output);
begin
  writeln('PASS ...6.4.3.5-4')
end.
```

This program simply tests that pointer types as described in the Pascal Standard are permitted.

```pascal
program t6p4p4dl(output);
type
  sett = set of 1..2;
  array = array[1..3] of integer;
  rekord = record
    a : integer;
    b : boolean
  end;
  ptr9 = *sett;
  pureptr = *pureptr;
var
  ptr1 : Integer;
  ptr2 : *real;
  ptr3 : *boolean;
  ptr4 : *sett;
  ptr5 : *array;
  ptr6 : *rekord;
  ptr7 : *pureptr;
  ptr8 : ptr9;
begin
  new(ptr1);
  new(ptr2);
  new(ptr3);
  new(ptr4);
  new(ptr5);
  new(ptr6);
  new(ptr7);
  new(ptr8);
  writeln('PASS ...6.4.4-1')
end.
```

This program tests the diagnostic that should be produced by the compiler if the type to which a pointer points is not found.

```pascal
program t6p4p4d2(output);
var
  pointer1 : ireal;
  pointer2 : irekord;
begin
  new(pointer1);
  pointer1:=nil;
  new(pointer2);
  pointer2:=nil;
  writeln('DEVIATES ...6.4.4-2')
end.
```

Pointers to items in the stack are not allowed, The T symbol is not FR,r';...to act as an operand giving the reference to a variable. The compiler deviates if the program compiler and prints DEVIATES.

```pascal
program t6p4p4d3(output);
var
  p: Integer;
  x:integer;
begin
  x:=10;
  p:=x;
  writeln('DEVIATES ...6.4.4-3, POINTER')
end.
```
The Pascal Standard states that types designated at two or more different places in the program text are identical if the same type identifier is used at these places, or if different identifiers are used which have been defined to be equivalent to each other. This program simply tests that the compiler conforms to the Standard's description of identity.

```
program t64p4p5dl(output);
type
tl = array[1..5] of boolean;
t2 = tl;
t3 = t2;

var
a : tl;
b : t2;
c : t3;

procedure identical(var a : tl; var b : t2; var c : t3);
begin
  a[1] := true;
b[1] := false;
c[1] := true;
end;

begin
  a[1] := true;
b[1] := false;
c[1] := false;
  identical(a, b, c);
end;
```

The compiler fails if the program does not compile.

[TEST 6.4.5-2, CLASS=DEVIANCE]

This program simply tests that the compiler does not deviate from the Standard in the case of subranges of the same host being treated as identical. The program should fail to compile/execute if the compiler conforms.

```
program t64p4p5d2(output);
type
  colour = (red, pink, orange, yellow, green, blue);
  subone = red .. yellow;
  subtwo = pink .. blue;

var
  colour1 : subone;
  colour2 : subtwo;

procedure test(var colour : subone);
begin
  writeln(' DEVIATES...6.4.5-2')
end;

begin
  colour2 := pink;
test(colour2)
end.
```

[TEST 6.4.5-3, CLASS=DEVIANCE]

This test is similar to 6.4.5-2, except that deviation in the case of arrays is tested. The program should not compile/execute if the compiler conforms.

```
program t64p4p5d3(output);
type
  arrayone = array[1..10] of char;
  arraytwo = array[1..10] of char;

var
  arrayone : arrayone;
  arraytwo : arraytwo;

procedure test(var array : arrayone);
begin
  writeln(' DEVIATES...6.4.5-3')
end;

begin
  arrayone := arraytwo;
test(arrayone)
end.
```
This program is similar to 6.4.5-3, except that deviance in the case of records is tested. The program should fail to compile/execute if the compiler conforms.

```pascal
program t6p4p5d4(output);
type
  recone = record
    a : integer;
    b : boolean
  end;
  rectwo = record
    c : integer;
    d : boolean
  end;
recordone : recone;
recordtwo : rectwo;

procedure test(var rec : recone);
begin
  writeln(' DEVIATES ••• 6.4.5-4')
end;

begin
  recordone := rectwo;
  test(recordone);
end.
```

Again, this test is similar to 6.4.5-4, except that deviance for pointers is tested. Although the two pointers in this example point to the same type, they are not identical. The compiler conforms if the program does not compile/execute.

```pascal
program t6p4p5d5(output);
type
  rekord = record
    a : integer;
  end;
  ptrone =rekord;
  ptrtwo = rekord;
  ptrtorec : ptrone;
  ptrtorectoo : ptrtwo;

procedure test(var ptr : ptrone);
begin
  writeln(' DEVIATES ••• 6.4.5-5')
end;

begin
  ptrtorectoo := nil;
  test(ptrtorectoo);
end.
```

Two types are compatible if they are identical or if one is a subrange of the other, or if both are subranges of the same type. This program tests these points, but with only subranges of the same type having some overlap. If the message produced is incomplete, or the program does not compile, then the compiler fails.

```pascal
program t6p4p5d6(output);
type
  colour = (red,pink,orange,yellow,green,blue,brown);
  colourtoo= colour;

  var
    coll : colour;
    col2 : colourtoo;
    subcoll : red..yellow;
    subcol2 : orange..blue;

begin
  if coll = col2 then writeln(' PA');
  if subcoll = subcol2 then writeln(' S...6.4.5-6')
end.
```
This program tests that two subranges of the same type with no overlap are considered as compatible by the compiler. The compiler fails if the program does not compile.

```pascal
program t6p4p5d7(output);
type
color = (red,pink,orange,yellow,green,blue,brown);
var
colI : red..yellow;
col2 : green..brown;
begin
colI:=yellow;
col2:=green;
if colI < col2 then writeln('PASS...6.4.5-7')
  else writeln('FAIL...6.4.5-7')
end.
```

The Pascal standard states that string types with the same number of components are compatible. The compiler fails if the program does not compile.

```pascal
program t6p4p5d8(output);
var
string1 : packed array[1..4] of char;
string2 : packed array[1..4] of char;
begin
string1:='ABCD';
string2:='EFGH';
if 'ABC' = 'ABC' then
  if string1 <> string2 then writeln('PASS...6.4.5-8')
  else writeln('FAIL...6.4.5-8')
else
  writeln('FAIL...6.4.5-8')
end.
```

The Pascal Standard states that set types of compatible base-types are compatible. This program tests that this is so for this compiler. The compiler conforms if the program does not compile.

```pascal
program t6p4p5d9(output);
type
colour = (red,pink,orange,yellow,green,blue,brown);
var
set1 : set of red..orange;
set2 : set of orange..brown;
begin
set1:=[orange];
set2:=[orange];
if set1=set2 then writeln('PASS...6.4.5-9')
  else writeln('FAIL...6.4.5-9')
end.
```

Some implementations may have an implicit ordering between different types, and allow these to be compared, etc., thus not conforming to the compatibility rules of the Pascal Standard. The compiler conforms if the program does not compile, or fails to run.

```pascal
program t6p4p5d10(output);
var
colour : (red,green,blue);
begin
if red < 0 then writeln('DEVIATES...6.4.5-10')
  else writeln('DEVIATES...6.4.5-10')
end.
```

The Pascal Standard permits compatibility only between string types of the same number of components. Some compilers may allow compatibility between string types with different numbers of components. The compiler conforms if the program does not compile.

```pascal
program t6p4p5d11(output);
begin
if 'CAT' < 'HOUND' then writeln('DEVIATES...6.4.5-11')
end.
```
If two types are declared equivalent, they inherit all properties in common, including operators and special attributes. This is checked by an analogue of type boolean. The compiler passes if the program compiles and prints PASS.

```pascal
program t6p4p5d12(output);
const
  on = true;
  off = false;
type
  logical = boolean;
  test : integer;
  bl, b2 : boolean;
l1, l2 : logical;
begin
  test := 0;
  bl := true;
  l1 := true;
  l2 := off;
  if l2 then test := test + 1;
  l2 := b2;
  if b1 = b2 then test := test + 1;
  b2 := l2;
  if b2 or l2 then test := test + 1;
  if test = 0 then
    writeln('PASS...6.4.5-12, TYPES')
  else
    writeln('FAIL...6.4.5-12, TYPES')
end.
```

This test is similar to 6.4.5-2, except that deviance in the case of arrays is tested. The program should not compile/execute if the compiler conforms.

```pascal
program t6p4p5d13(output);
type
 Jarurrayone = array[1..10] of boolean;
  Jararraytwo = array[1..10] of boolean;
var
  arrayone : Jararrayone;
  arraytwo : Jararraytwo;
procedure test(var array : arrayone);
begin
  writeln('DEVIATES...6.4.5-13')
end;
```

The two array types, arrayone and arraytwo, are not identical and hence the call to TEST should fail.

```pascal
procedure test(arraytwo);
begin
  writeln('PASS...6.4.5-13')
end.
```

This program tests that all assignment compatible types as described by the Pascal Standard, are permitted by this compiler. This program tests only those uses in assignment statements. All cases have been tested elsewhere, but are included here together for consistency. The compiler fails if one or more of the cases below are rejected.

```pascal
program t6p4p6d1(output);
type
  colour = (red, pink, yellow);
  rekord = record
    a : integer;
    b : boolean
  end;
var
  i, j : integer;
  co1, co2, co3 : colour;
  record1 = rekord;
  record2 = rekord;
begin
  i := 2;
  j := 1;
  co1 := red;
  co2 := pink;
  co3 := yellow;
  record1 := rekord;
  record2 := rekord;
end.
```

This program tests that all assignment compatible types as described by the Pascal Standard, are permitted by this compiler. This program tests only those uses in assignment statements. All cases have been tested elsewhere, but are included here together for consistency. The compiler fails if one or more of the cases below are rejected.

```pascal
program t6p4p6d1(output);
type
class performance
program t6p4p6d1(output);
type
class performance
```
This test is similar to 6.4.6-1, except that it tests the use of assignment compatibility in actual and formal parameters. The compiler fails if the program does not compile.

program t6p4p6d2(output);
type
colour = (red,pink,yellow,green);
subcoll = yellow,green;
subcol2 = set of colour;
subcol3 = set of pink,green;
var
a : integer;
b : real;
colour1 : colour;
colour2 : pink,green;
colour3 : set of colour;
colour4 : set of yellow,green;
begin
compat(2,2.4,yellow,yellow,[pink],[pink]);
a:=2;
b:=3.1;
colour1:=pink;
colour2:=green;
colour3:=[yellow];
colour4:=[yellow];
compat(a,b,colour1,colour2,colour3,colour4);
compat(a,a,colour2,colour2,colour4,colour4);
writeln(' PASS...6.4.6-2');
end.

This program tests a part of 6.5.2.1, that states that an index expression is assignment compatible with the index type specified in the definition of the array type. The compiler fails if the program does not compile.

program t6p4p6d3(output),
type
colour = (red,pink,orange,yellow,green),
intensity = (bright,dull);
var
array1 : array[yellow..green] of boolean,
array2 : array[colour] of intensity,
array3 : array[1..99] of integer;
colour1 : red..yellow;
i : integer;
begin
array1[yellow]:=true;
colour1:=yellow,
array1[colour1]:=false;
array2[colour1]:=bright,
array3[1]:=0,
i:=2;
array3[i*3+2]:=1;
writeln(' PASS...6.4.6-3');
end.

The Pascal standard says that if the two types in an assignment compatibility test (T1 and T2) are compatible ordinal types and the value of the expression E which is of type T2 is not in the closed interval specified by the type T1, an error occurs. Does this compiler detect this?

program t6p4p6d4(output);
type
 subrange = 0..5;
var
  i : subrange;
begin
  i:=5;
  writeln(' ERROR NOT DETECTED...6.4.6-4');
end.
This program is similar to 6.4.6-4, except that parameter assignment compatibility is tested. The program causes an error to occur which should be detected.

```pascal
program t6p4p6d5(output);
  type
    subrange = 0..5;
  var
    i : subrange;
  procedure test(a subrange);
  begin
    a:=5
  end;
begin
  i:=5;
  test(i*2);  (* error *)
  writeln(' ERROR NOT DETECTED...6.4.6-5')
end.
```

This program is similar to 6.4.6-4, except that array subscript assignment compatibility is tested. The program causes an error which should be detected.

```pascal
program t6p4p6d6(output);
  type
colour = (red,pink,orange,yellow,green);
  var
    setone set of subone;
    settwo set of subtwo;
  begin
    settwo:=[pink,yellow];
    setone:=settwo;
    writeln(' ERROR NOT DETECTED...6.4.6-6')
  end.
```

Similarly for 5.4.6-4, if two types are compatible set types, and any of the members of the set expression E (of type T2) is not in the closed interval specified by the base-type of the type T1, an error occurs. Again, does the compiler detect this.

```pascal
program t6p4p6d7(output);
  type
colour = (red,pink,orange,yellow,green,blue);
  subone = red..orange;
  subtwo = pink..yellow;
  var
    setone : set of subone;
    settwo : set of subtwo;
  begin
    settwo:=[pink,yellow];
    setone:=settwo;
    writeln(' ERROR NOT DETECTED...6.4.6-7')
  end.
```

This test is similar to 6.4.6-7, except that assignment compatibility for sets passed as parameters is tested. The program causes an error which should be detected.

```pascal
program t6p4p6d8(output);
  type
colour = (red,pink,orange,yellow,green,blue);
  subone = red..green;
  subtwo = set of yellow..blue;
  var
    setone : set of subone;
    procedure test(a : settwo);
  begin
    test(setone);
    writeln(' ERROR NOT DETECTED...6.4.6-8')
  end.
```
The Pascal Standard allows assignment of integers to reals, but not reals to integers. Does this compiler allow assignment of reals to integers. If so, it does not conform to the Standard. The compiler conforms if the program does not compile.

```pascal
program t6p4p6d9(output);
var
  i : real;
  j : integer;
procedure test(a:integer);
begin
end;
begin
  i := 6.345;
  j := i;
  test(6.345);
  writeln(' DEVIATES ...6.4.6-9')
end.
```

The Pascal Standard states that the two types T1 and T2 (in determining assignment compatibility) must neither be a file type nor a structured type with a file component. This program tests the first part of this statement. The compiler conforms if the program does not compile.

```pascal
program t6p4p6d10(output);
var
  file1 : text;
  file2 : text;
begin
  reset(file1);
  rewrite(file2);
  writeln(file1,'ABC');
  file2:=file1;
  writeln(' DEVIATES ...6.4.6-10')
end.
```

The standard specifies that a filetype T2 cannot be assignment-compatible with an identical type T1, nor can a structure containing such a filetype. This precludes any assignments involving files. The compiler deviates if the program compiles and prints DEVIATES.

```pascal
program t6p4p6d11(output);
var
  file1, file2 : text;
begin
  rewrite(file1);
  writeln(file1,' DEVIATES');
  file2:=file1;
  writeln(' DEVIATES ...6.4.6-12, FILES');
  f2:=f1;
end.
```
Here is included two examples from the Pascal Standard. The first is from section 6.4.7, and consists of legal type declarations. The second is from section 6.5.1, and consists of legal variable declarations. The compiler fails if the program does not compile.

\begin{verbatim}
var i10, i11 := 0; i12 := 2; i13 := 3; i14 := 4; i15 := 5; i16 := 6; i17 := 7; i18 := 8; i19 := 9; i20 := 10; i21 := 11; i22 := 12; i23 := 13; i24 := 14; i25 := 15; i26 := 16; i27 := 17; i28 := 18; i29 := 19; i30 := 20; i31 := 21; i32 := 22; i33 := 23; i34 := 24; i35 := 25; i36 := 26; i37 := 27; i38 := 28; i39 := 29; i40 := 30; i41 := 31; i42 := 32; i43 := 33; i44 := 34; i45 := 35; i46 := 36; i47 := 37; i48 := 38; i49 := 39; i50 := 40; i51 := 41; i52 := 42; i53 := 43; i54 := 44; i55 := 45; i56 := 46; i57 := 47; i58 := 48; i59 := 49; i60 := 50; i61 := 51; i62 := 52; i63 := 53; i64 := 54; i65 := 55; i66 := 56; i67 := 57; i68 := 58; i69 := 59; i70 := 60; i71 := 61; i72 := 62; i73 := 63; i74 := 64; i75 := 65; i76 := 66; i77 := 67; i78 := 68; i79 := 69; i80 := 70; i81 := 71; i82 := 72; i83 := 73; i84 := 74; i85 := 75; i86 := 76; i87 := 77; i88 := 78; i89 := 79; i90 := 80; i91 := 81; i92 := 82; i93 := 83; i94 := 84; i95 := 85; i96 := 86; i97 := 87; i98 := 88; i99 := 89; i100 := 90; i101 := 91; i102 := 92; i103 := 93; i104 := 94; i105 := 95; i106 := 96; i107 := 97; i108 := 98; i109 := 99; i110 := 100; i111 := 101; i112 := 102; i113 := 103; i114 := 104; i115 := 105; i116 := 106; i117 := 107; i118 := 108; i119 := 109; i120 := 110; i121 := 111; i122 := 112; i123 := 113; i124 := 114; i125 := 115; i126 := 116; i127 := 117; i128 := 118; i129 := 119; i130 := 120; i131 := 121; i132 := 122; i133 := 123; i134 := 124; i135 := 125; i136 := 126; i137 := 127; i138 := 128; i139 := 129; i140 := 130; i141 := 131; i142 := 132; i143 := 133; i144 := 134; i145 := 135; i146 := 136; i147 := 137; i148 := 138; i149 := 139; i150 := 140; i151 := 141; i152 := 142; i153 := 143; i154 := 144; i155 := 145; i156 := 146; i157 := 147; i158 := 148; i159 := 149; i160 := 150; i161 := 151; i162 := 152; i163 := 153; i164 := 154; i165 := 155; i166 := 156; i167 := 157; i168 := 158; i169 := 159; i170 := 160; i171 := 161; i172 := 162; i173 := 163; i174 := 164; i175 := 165; i176 := 166; i177 := 167; i178 := 168; i179 := 169; i180 := 170; i181 := 171; i182 := 172; i183 := 173; i184 := 174; i185 := 175; i186 := 176; i187 := 177; i188 := 178; i189 := 179; i190 := 180; i191 := 181; i192 := 182; i193 := 183; i194 := 184; i195 := 185; i196 := 186; i197 := 187; i198 := 188; i199 := 189; i200 := 190; i201 := 191; i202 := 192; i203 := 193; i204 := 194; i205 := 195; i206 := 196; i207 := 197; i208 := 198; i209 := 199; i210 := 200;
end.
\end{verbatim}
This test is similar to 6.4.5-6, except that a two-dimensional array is used. This may present some problems to particular implementations.

```pascal
program t6p5p3p2d1(output);
var
  a: array[1..10,1..10] of integer;
  i: integer;
begin
  i:=3;
  a[i*2,i*4]:=0;
  writeln(' ERROR NOT DETECTED...6.5.3.2-1')
end.
```

This test checks that the two ways of indexing a multi-dimensional array are equivalent. The compiler fails if the program does not compile and print PASS.

```pascal
program t6p5p3p2d2(output);
var
  a: array[1..4,1..4] of integer;
  p: packed array[1..4,1..4] of char;
  q: packed array[1..4] of packed array[1..4] of char;
  i,j,counter:integer;
begin
  counter:=0;
  for i:=1 to 4 do
    for j:=1 to 4 do
      begin
        a[i,j] := j;
        b[i,j] := j;
        case j of
          1:
            begin
              p[i,j] := 'P';
              q[i,j] := 'P';
            end;
          2:
            begin
              p[i,j] := 'A';
              q[i,j] := 'A';
            end;
          3:
            begin
              p[i,j] := 'I';
              q[i,j] := 'I';
            end;
          4:
            begin
              p[i,j] := 'L';
              q[i,j] := 'L';
            end;
        end;
      end;
  for i:=1 to 4 do
    for j:=1 to 4 do
      begin
        if a[i][j] <> a[i,j] then
          counter:=counter+1;
        if b[i][j] <> b[i,j] then
          counter:=counter+1;
        if p[i][j] <> p[i,j] then
          counter:=counter+1;
        if q[i][j] <> q[i,j] then
          counter:=counter+1;
      end;
if counter=0 then
  writeln(' PASS...6.5.3.2-2, INDEXING')
else
  writeln(' FAIL...6.5.3.2-2, INDEXING')
end.
```
The Pascal Standard states that the existence of a file variable `f` with components of type `T` implies the existence of a buffer variable of type `T`.

Only the one component of a file variable determined by the current file position is directly accessible. The program tests that file buffers may be referenced in this implementation.

The compiler fails if the program does not compile.

```pascal
program t6p5p3p4dl(output);
type
  rekord = record
    a : integer;
    b : real
  end;
var
  fyle : file of rekord;
begin
  rewrite(fyle);
  fyleT.urray[1] := '0';
  fyleT.a := 10;
  fyleT.b := 2.345;
  put(fyle);
  with fyle do begin
    ury[1] := '0';
    a := 4;
    b := 3.456
  end;
  put(fyle);
  writeln(' PASS ... 6.5.3.4-1')
end.
```

[TEST 6.5.3.4-1, CLASS=CONFORMANCE]

The Pascal Standard states that an error occurs if a pointer variable has a value NIL at the time it is dereferenced. This program tests that the error is detected. The diagnostic should be checked for suitability.

```pascal
program t6p5p4dl(output);
type
  rekord = record
    a : integer;
    b : boolean
  end;
var
  pointer : rekord;
begin
  pointer := nil;
  pointerT.a := 1;
  pointerT.b := true;
  writeln(' ERROR NOT DETECTED ... 6.5.4-1')
end.
```

[TEST 6.5.4-1, CLASS=ERRORHANDLING]

Similarly to 6.5.4-1, an error occurs if a pointer variable has an undefined value when it is dereferenced.

```pascal
program t6p5p4d2(output);
type
  rekord = record
    a : integer;
    b : boolean
  end;
var
  pointer : rekord;
begin
  pointerT.a := 1;
  pointerT.b := true;
  writeln(' ERROR NOT DETECTED ... 6.5.4-2')
end.
```

[TEST 6.5.4-2, CLASS=ERRORHANDLING]
This program simply tests the syntax for procedures as defined by the Pascal Standard. The compiler fails if the program does not compile.

program t6p6p1d1(output);
var
  a : integer;
  b : real;
procedure withparameters(g : integer; h : real);
var
c : integer;
d : real;
begin
  c:=g;
d:=h
end;
procedure parameterless;
begin
  write(' PASS')
end;
begin
  a:=1;
b:=2;
withparameters(a,b);
  parameterless;
  writeln('...6.6.1-1')
end.

This program tests the implementation of FORWARD declaration, recursive activation, and multilevel referencing of a var parameter in procedures. The compiler fails if the program will not compile.

program t6p6p1d2(output);
var
c : integer;
procedure one(var a : integer);
forward;
procedure two(var b : integer);
begin
  b:=b+1;
one(b)
end;
procedure one;
begin
  a:=a+1;
  if a = 1 then two(a)
end;
begin
  c:=0;
one(c);
  if c = 3 then
    writeln(' PASS...6.6.1-2')
end.
This program tests if the compiler allows the formal parameter list to be included in the subsequent procedure declaration of a forward procedure. The compiler conforms to the Standard if the program does not compile.

program t6p5pld4(output);
var
  c : integer;
procedure one(var a : integer);
forward;
procedure two(var b : integer);
begin
  b:=b+1;
  one(b)
end;
procedure one(var a : integer);
begin
  a:=a+1;
  if a = 1 then two(a)
end;
begin
  c:=0;
  one(c);
  writeln('DEVIATES...6.6.1-4')
end.

[TEST 6.6.1-5, CLASS=DEVIANCE]

{ If the compiler permits the formal parameter list to be included in the subsequent procedure declaration of a forward procedure (6.6.1-4), does it check the parameter list is the same? The compiler deviates if the program compiles, and only conforms if the second formal parameter list is flagged as an error. }

program t6p5pld5(output);
var
  c : integer;
procedure one(var a : integer);
forward;
procedure two(var b : integer);
begin
  b:=b+1;
  one(b)
end;
procedure one(var a : integer);
begin
  a:=a+1;
  if a = 1 then two(a)
end;
begin
  c:=0;
  one(c);
  writeln('DEVIATES...6.6.1-5')
end.

[TEST 6.6.1-6, CLASS=DEVIANCE]

{ This program tests the compilers actions for a procedure declared as forward, but no matching subsequent procedure declaration for the forward procedure occurs. }

program t6p6pld6(output);
var
  c : integer;
procedure two(var b : integer);
forward;
procedure one(var a : integer);
begin
  a:=a+1;
  if a = 1 then two(a)
end;
begin
  c:=0;
  one(c);
  writeln('DEVIATES...6.6.1-6, FORWARD PROCEDURE')
end.
This test checks that procedures may be nested to 15 levels. The test may detect a small compiler limit. The limit may arise due to failure of a register allocation scheme, a limited reserved size for a display, or a field set aside for lexical level information, or some combination of these.

```pascal
program t5p6p1d7(output);
var
  i: integer;
procedure p1;
  procedure p2;
    procedure p3;
      procedure p4;
        procedure p5;
          procedure p6;
            procedure p7;
              procedure p8;
                procedure p9;
                  procedure p10;
                    procedure p11;
                      procedure p12;
                        procedure p13;
                          procedure p14;
                            procedure p15;
                              begin
                                i := i + 1;
                              end;
                            begin
                              p15
                            end;
                          begin
                          p14
                        end;
                    begin
                      p13
                    end;
                begin
                  p12
                end;
              begin
                p11
              end;
            begin
              p10
            end;
          begin
            p9
          end;
        begin
          p8
        end;
      begin
      p7
    end;
  begin
    p6
  end;
begin
  i := 0;
begin
  p1;
writeln('NESTED PROCEDURES TO 15 LEVELS IMPLEMENTED...6.6.1-7');
end.
```
This program simply tests the syntax for functions as defined by the Pascal Standard. The compiler fails if the program does not compile.

```pascal
program t6p6p2dl(output);
var
  a, twopisquared : real;
  b : integer;
function power(x : real; y integer) :real; { y>=0 }
var
  w,z : real;
  i : integer;
begin
  w:=x;
  z:=1;
  i:=y;
  while i > 0 do
  begin
    [ z*(w tothepower i)=x tothepower y ]
    if odd(i) then z:=z*w;
    i:=i div 2;
    w:=sqr(w)
  end;
  ( z=x tothepower y }
  power:=z
end;
function twopi : real;
begin
twopi:=6.283185
end;
begin
  a:=twopi;
  b:=2;
  twopisquared:=power(a,b);
  writeln('PASS...6.6.2-1')
end.
```

Similarly to 6.6.2-2, functions may be declared as forward. This program tests that forward declaration and recursion in functions is permitted. The compiler fails if the program does not compile.

```pascal
program t6p6p2d2(output);
var
  c : integer;
function one(a integer)
forward;
function two(b integer) :integer;
begin
  x:=b+1;
  x:=one(x);
  two:=x
end;
function one;
begin
  y:=a+l;
  if y=l then y:=two(y);
  one:=y
end;
begin
  c:=O;
  c:=one(c);
  if c = 3 then
    writeln('PASS...6.6.2-2')
end.
```
The Pascal Standard specifies that the result type of a function can only be a simple type or a pointer type.

This program checks that the simple types and pointer types are permitted. The compiler fails if the program does not compile.

```pascal
program t6p6p2d3(output);
type
  subrange = 0..3;
  enumerated = (red,yellow,green);
  rectype = record
    a : integer
  end;
  ptrtype = rectype;
var
  a : real;
  b : integer;
  c : boolean;
  d : subrange;
  e : enumerated;
  f : char;
  g : ptrtype;
function one : real;
begin
  one:=2.63
end;
function two : integer;
begin
  two:=2
end;
function three : boolean;
begin
  three:=false
end;
function four : subrange;
begin
  four:=2
end;
function five : enumerated;
begin
  five:=yellow
end;
function six : char;
begin
  six:='6'
end;
function seven : ptrtype;
begin
  seven:=nil
end;
begin
  a:=one;
  b:=two;
  c:=three;
  d:=four;
  e:=five;
  f:=six;
g:=seven;
  writeln(' PASS...6.6.2-3')
end.
```

This program tests the compiler's actions when the type of result returned by a function is not a simple type. All the cases should be rejected by the compiler if it conforms to the Standard.

```pascal
program t6p6p2d4(output);
type
  wrekord = record
    a : integer;
    b : boolean
  end;
  sett = set of 0..3;
  urray = array[1..3] of char;
var
  record1 : wrekord;
  setl : sett;
  array1 : urray;
begin
  record1:=one;
  setl:=two;
  array1:=three;
  writeln(' DEVIATES...6.6.2-4')
end.
```
The Pascal Standard specifies that at least one assignment statement which assigns a value to the function identifier must occur in the function block. Does the compiler permit a function declaration with no assignment to the function identifier? The compiler deviates if it does.

program t5p6p2d5(output);
var
 a : integer;
function illegal(var b : integer)
var
 x : integer;
begin
 x:=b*2
end;
begin
 a:=2;
a:=illegal (a);
writeln(' DEVIATES...6.6.2-5')
end.

This test checks that functions are not prohibited from altering their environment (ie. side effects). Though side effects are generally not to be encouraged, they are part of standard Pascal and do have genuine uses. Functions with side effect occur elsewhere in the validation suite.

program t5p6p2d7(output);
type
 ptrtochar = fchar;
var
 c1,c2,c3,dummy:char;
p1,p2:ptrtochar;
function testa(ptr:ptrtochar):char;
{ sneakiest, uses pointers }
var
 pp:ptrtochar;
begin
 pp:=ptr;
 ppT:='P';
testa:='l'
end;
procedure assign;
{ used by testb }
begin
 cl:='i'
end;
function testb:char;
{ sneaky, calls a procedure }
begin
 assign;
testb:='2'
end;
function testc:char;
{ blatantly changes the environment via write }
begin
write('x',pIT,c1,c2,c3,p2f);
testc:='6'
end;
function testd:ptrtochar;
{ blatantly sneaky: modifying the environment via new and then passing it out }
var
 pp:ptrtochar;
begin
 new(pp);
 ppT:='D';
testd:=pp
end;
function teste:char;
{ the most used side effect:global access }
begin
c2:='S';
teste:='3'
end.

program t5p6p2d6(output);
var
 radius ,
circlearea : real;
function area(a : real) real;
var
 x : real;
begin
 if a > 0 then x:=3.1415926*a*a
 else area:=0
end;
begin
 radius:=2;
circlearea:=area(radius);
writeln(' ERROR NOT DETECTED...6.6.2-6')
end.
function testf(var c:char):char;
[straightforward]
begin
  c:='S';
  testf:~'4'
end;

begin {of main program}
  new(pl);
  pi:='F'; cl:='A'; c2:='I'; c3:='L';
  p2:~nil;
  dummy:~testa(pl);
  dummy:~testb;
  dummy:~teste;
  dummy:~testf(c3);
  p2:~testd;
  dummy:=testc;
  writeln('...6.6.2-7, ENVIRONMENT')
end.
colone, coltwo, colthree, v, w, x, y, z, ptr;

testtwo(a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, ptr);

end.

[TEST 6.6.3.1-2, CLASS=CONFORMANCE]

[ This program is similar to 6.6.3.1-1, except that set, record and array parameter lists are tested.
The compiler fails if the program does not compile. ]

program test6p6p3p1d2(output);
begin
end.

{TEST '!'he
This program is similar to 6.6.3.1-1, except that set, record and array parameter lists are tested.

procedure testone(set1, set2, set3, set4, set5, set6: set;

record one, two, three, four, five, six: record;
array[1..20] of boolean;

var
setone, settwo, setthree, setfour, setfive, setsix: set;
rec1, rec2, rec3, rec4, rec5: record;
arrayone, arraytwo, arraythree, arrayfour: array;

procedure testone(set1, set2, set3, set4, set5, set6: set;

record one, two, three, four, five, six: record;
array[1..20] of boolean;

var
setone, settwo, setthree, setfour, setfive, setsix: set;
rec1, rec2, rec3, rec4, rec5: record;
arrayone, arraytwo, arraythree, arrayfour: array;

begin
write(' PASS ');
end.

procedure testtwo(var set1, set2, set3, set4, set5, set6: set;

record one, two, three, four, five, six: record;
array[1..20] of boolean;

var
setone, settwo, setthree, setfour, setfive, setsix: set;
rec1, rec2, rec3, rec4, rec5: record;
arrayone, arraytwo, arraythree, arrayfour: array;

begin
write('... 6.6.3.1-2');
end.

begin
setone := [1]; settwo := [1]; setthree := [1];
setfour := [1]; setfive := [1]; setsix := [1];
rec1 := 1; rec2 := 1; rec3 := 1;
recfour := 1; recfive := 1;
arrayone[true] := false; arraytwo[true] := false;
arraythree[true] := false; arrayfour[true] := false;

testone(setone, settwo, setthree, setfour, setfive, setsix,
rec1, rec2, rec3, rec4, rec5,
arrayone, arraytwo, arraythree, arrayfour);

testtwo(setone, settwo, setthree, setfour, setfive, setsix,
rec1, rec2, rec3, rec4, rec5,
arrayone, arraytwo, arraythree, arrayfour);
end.

[TEST 6.6.3.1-3, CLASS=CONFORMANCE]

[ This program tests that files may be passed to
procedures as parameters, as a file is a type, and any
type may be passed as a parameter.
The compiler fails if the program does not compile.
]

program test6p6p3p1d3(output);
type

type

file = text;

var
e1y = file;

procedure test(file anyfile: file);

begin
rewrite(anyfile);
write(' THIS FILE WAS A PARAMETER');
write(' PASS... 6.6.3.1-3');
end;

begin
write(file);
end.

procedure test6p6p3p1d4(output);

var

text file;

begin
rewrite(file);
write(file, 'THIS FILE WAS A PARAMETER');
write(' PASS... 6.6.3.1-4');
end;

begin
write(file);
end.

{TEST 6.6.3.1-4, CLASS=DEVIANCE}

[ The occurrence of an identifier within an Identifier list of
a parameter group is its defining occurrence as a parameter
Identifier for the formal parameter list in which it occurs
and any corresponding procedure block or function block.
This precludes the declaration of a local variable with the same
name as an identifier in the formal parameter list.
Does the compiler detect this as an error, or allow it
to occur with some form of side effect?
The compiler conforms if the program does not compile.
]

program test6p6p3p1d4(output);

var

1: integer;

procedure deviates(var x: integer);

begin
x := 2 * x;
write(' DEVIATES... 6.6.3.1-4: x=', x);
end;

procedure deviates(x: integer);

begin
x := 0;
x := 2 * x;
write(' DEVIATES... 6.6.3.1-4: x=', x);
end;

begin
i := 5;
deviates(i);
i := 5;
deviates(i);
end.
When a procedure (or function) with a parameter list is included in the formal parameter list of another procedure (or function), the identifiers in the parameter list of the procedure parameter have defining occurrences for that list and the corresponding block for the procedure only, and not for the block of the procedure to which it is passed.

The example in this program should be passed by the compiler.

program t6p6p3p0d5(output);
var
i : integer;
procedure alsoconforms(x : integer);
begin
write1n(' PASS...6.6.3.1-5')
end;
procedure conforms(procedure alsoconforms(x : integer));
var x : boolean;
begin
x:=true;
alsoconforms(i)
end;

begin
i:=2;
conforms(alsoconforms(i))
end.

program t6p6p3p3d2(output);
var
direct : integer;
pass : boolean;
procedure indirection(var indirect integer; var result boolean);
begin
indirect:=2;
if indirect<>direct then
result:=false
else
result:=true
end;
begin
direct :=1,
pass:=false;
indirection(direct,pass);
if pass then
write1n(' PASS...6.6.3.3-2')
else
write1n(' FAIL...6.6.3.3-2')
end.

The Pascal Standard states that any operation involving the formal parameter is performed immediately on the actual parameter. Depending on how variable parameter passing is implemented, this test may cause some compilers to fail.

The compiler fails if the program does not compile, or the program states that this is so.

program t6p6p3p3d2(output);
var
direct : integer;
pass : boolean;
procedure indirection(var indirect integer; var result boolean);
begin
indirect:=2;
if indirect<>direct then
result:=false
else
result:=true
end;
begin
direct :=1;
pass:=false;
indirection(direct,pass);
if pass then
write1n(' PASS...6.6.3.3-2')
else
write1n(' FAIL...6.6.3.3-2')
end.

This program would have tested that the actual parameters to a procedure/function are assignment compatible with the type of the formal parameter. Depending on how variable parameter passing is implemented, this test may cause some compilers to fail. The compiler fails if the program does not compile, or the program states that this is so.
If the variable passed as a parameter involves the indexing of an array, or the dereferencing of a pointer, then these actions are executed before the activation of the block. The compiler fails if the program does not compile or the program states that this is so.

```pascal
program t6p6p3p3d3(output);
{ TEST 6.6.3.3-3, CLASS=CONFORMANCE }
type
  rekord = record
    a: integer;
    link:rekord;
    back:rekord
  end;
var
  arrray:array[1..2] of integer;
i:integer;
tempotr,ptr:rekord;
procedure call(arraylocn:integer; ptrderef:integer);
begin
  i:=i+1;
  ptr:=ptrT.link;
  if (array[i-1]<>arraylocn) or
    (ptrT.backT.a<>ptrderef) then
    writeln(' FAIL...6.6.3.3-3')
  else
    writeln(' PASS...6.6.3.3-3')
end;
begin
  arrray[1]:=1;
  arrray[2]:=2;
i:=1;
naw(ptr);
ptrT.a:=1;
naw(tempotr);
tempotrT.a:=2;
ptrT.link:=temptr;
tempotrT.back:=ptr;
call(array[i],ptrT.a)
end.
```

This program tests that procedures may be passed to other procedures and functions as parameters. The compiler fails if the program does not compile and run.

```pascal
program t6p6p3p4d1(output);
{ TEST 6.6.3.4-1, CLASS=CONFORMANCE }
var
  i:integer;
 procedure a(procedure b);
begin
  write(' PASS');
b
end;
procedure c;
begin
  write(' .')
end;
function d(procedure b)
begin
  b;
d:=2
end;
begin
  a(c);
i:=d(c);
  if i=2 then
    writeln('.6.6.3.4-1')
end.
```
This program tests that the environment of procedure parameters is as stated in the Pascal Standard. The compiler fails if the program does not compile, or the program states that this is so.

```
program tfip6p3p4d2(output);
var
  globalone, globaltwo : integer;
procedure p(procedure f(procedure a,b), procedure g);
  var
    localtop : integer;
  procedure r;
  begin
    if globalone = 1 then
      if (globaltwo > 2) or (localtop > 1) then
        writeln(' FAIL1...6.6.3.4-2')
      else
        writeln(' PASS...6.6.3.4-2')
    else
      writeln(' FAIL2...6.6.3.4-2')
  end;
begin
  globaltwo := globalone + 1;
  localtop := globaltwo;
  if globaltwo = 1 then
    p(f, r)
  else
    begin
      if globaltwo = 1 then
        f(r) 
      else
        f(g, r)
    end;
  procedure q(procedure f, g);
  begin
    if f then
      r
  end;
procedure dummy;
begin
  writeln(' FAIL4...6.6.3.4-2')
end;
begin
  globalone := 1;
  globaltwo := 0;
  p(q, dummy)
end.
```

Similarly to 6.6.3-1, this program tests that functions may be passed to procedures and functions as parameters. The compiler fails if the program does not compile and runs.

```
program tfip6p3p5d1(output);
var
  j : integer;
procedure a(function b : integer);
  var
    i : integer;
  begin
    i := b;
    write(' PASS')
  end;
function c : integer;
begin
  c := 2
end;
function d (function b : integer)
begin
  d := b
end;
begin
  a(c)
  j := d(c);
  if j = 2 then
    writeln(' DEVIATES...6.6.3.5-1')
end.
```

This test checks functional compatibility in that function types are required to be identical. The compiler deviates if the program compiles and prints 'DEVIATES.'
This test checks that constants are not permitted as var parameters. The compiler deviates if the program compiles and prints DEVIATES.

```pascal
program t6p5p3p6d1(output);
const
  x=1;
var
  y:integer;
procedure assign(var p:integer);
begin
  p:=100
end;
begin
  assign(y);
  assign(x); {disallowed}
  writeln(' DEVIATES ... 6.6.3.6-1, VAA PARAMS')
end.
```

This test checks that parameter list compatibility is correctly implemented. The compiler deviates if the program compiles and prints DEVIATES.

```pascal
program t6p5p3p6d2(output);
type
  natural = 0..maxint;
procedure actual(i:integer; n:natural);
begin
  i:=n
end;
procedure p(procedure formal(var a:integer;b:integer));
var
  k,l: integer;
begin
  k:=1; 1:=2;
  formal(k,l)
end;
begin
  p(actual);
  writeln(' DEVIATES ... 6.6.3.6-2, VALUE PARS NOT IDENT TYPES')
end.
```

This test checks that parameter list compatibility is correctly implemented. The compiler deviates if the program compiles and prints DEVIATES.

```pascal
program t6p5p3p6d3(output);
type
  natural = 0..maxint;
procedure actual(i:integer; n:natural);
begin
  i:=n
end;
procedure p(procedure formal(var a:integer;b:natural));
var
  k,l:integer;
begin
  k:=1; 1:=2;
  formal(k,l)
end;
begin
  p(actual);
  writeln(' DEVIATES ... 6.6.3.6-3, VALUE/VAR MISMATCH')
end.
```

This test checks that parameter list compatibility is correctly implemented. The compiler deviates if the program compiles and prints DEVIATES.

```pascal
program t6p5p3p6d4(output);
type
  natural = 0..maxint;
procedure actual(var i:integer;var n:natural);
begin
  i:=n
end;
procedure p(procedure formal(var a:integer;var b:integer));
var
  k,l: integer;
begin
  k:=1; 1:=2;
  formal(k,l)
end;
begin
  p(actual);
  writeln(' DEVIATES ... 6.6.3.6-4, VAR PARS NOT IDENT TYPES')
end.
```
This test checks that parameter list compatibility is correctly implemented. The compiler deviates if the program complies and prints DEVIATES.

```pascal
program t6p6p3p6d5(output);
type
  natural = 0 .. maxint;
procedure actual(i:integer; j:integer; n:natural);
begin
  i:=n
end;
procedure p(procedure formal(a:integer;b:integer);
  var
  k,l:integer;
begin
  k:=l; l:=2;
  formal(k,l)
end;
begin
  p(actual) ;
  writeln(' DEVIATES...6.6.3.6-5, NO OF PARS DIFFERENT')
end.
```

This program tests that predefined standard procedures may be redefined with no conflict. The compiler fails if the program does not compile and run.

```pascal
program t6p6p4p1dl(output);
var
  i : integer;
procedure write(var a : integer);
begin
  a:=a+2
end;
procedure get(var a : integer);
begin
  a:=a*2
end;
begin
  i:=0;
  write(i);
  get(i);
  if i=4 then
    writeln(' PASS...6.6.4.1-1')
  else
    writeln(' FAIL...6.6.4.1-1')
end.
```

This program causes an error to occur, as eof(f) does not yield true prior to execution of a put on the file f. The error should be detected at compile-time or run-time.

```pascal
program t6p6p5p2d1(output);
var
  fyle : text;
begin
  rewrite(fyle);
  writeln(fyle,'ABC');
  reset(fyle); { eof is false and fT='A'
  put(fyle) ; { causes an error }
  writeln(' ERROR NOT DETECTED...6.6.5.2-1')
end.
```

This program causes an error to occur as eof(f) does not yield false prior to execution of a get on the file f. The error should be detected at compile-time or run-time.

```pascal
program t6p6p5p2d2(output) ;
var
  fyle : text;
begin
  rewrite( fyle);
  writeln( Eyle, 'ABC');
  reset (fyle) ;
  get (fyle) ; { fyleT= 'B' } 
  get (fyle) ; { fyleT= 'C' } 
  get(fyle); { fyleT= undefined...eof is true }
  get(fyle); { error since eof is true }
  writeln(' ERROR NOT DETECTED...6.6.5.2-2')
end.
```

This program tests if true is assigned to eof if the file f is empty when reset.

```pascal
program t6p6p5p2d3(output);
var
  fyle : text;
begin
  reset(fyle);
  if eof(fyle) then
    writeln(' PASS...6.6.5.2-3')
  else
    writeln(' FAIL...6.6.5.2-3')
end.
```
This program tests that the first element of a file is assigned to the buffer variable when the procedure reset is used with the file.

```pascal
program t56p5p2d4(output);
var
  fyle : text;
begin
  rewrite(fyle);
  writeln(fyle,'ABC');
  writeln(fyle,'DEF');
  reset(fyle);
  if fyle^='A' then
    writeln(' PASS...6.6.5.2-4')
  else
    writeln(' FAIL...6.6.5.2-4')
end.
```

This program checks that a rewrite on the file sets eof to be true.

```pascal
program t56p5p2d5(output);
var
  fyle: text;
begin
  rewrite(fyle);
  if eof(fyle) then
    writeln(' PASS...6.6.5.2-5')
  else
    writeln(' FAIL...6.6.5.2-5')
end.
```

This program causes an error to occur by changing the current file position of a file, while the buffer variable is an actual parameter to a procedure. The error should be detected by the compiler, or at run-time.

```pascal
program t56p5p2d6(output);
var
  fyle: text;
procedure naughty(f : char);
begin
  if f='G' then
    put(fyle)
end;
begin
  rewrite(fyle);
  fyle^:='G';
  naughty(fyle);
  writeln(' ERROR NOT DETECTED...6.6.5.2-6')
end.
```

This test is similar to 6.6.5.2-6, except that the buffer variable is an element of the record variable list of a with statement. The error should be detected by the compiler or at run-time.

```pascal
program t56p5p2d7(output);
type
  sex = (male,female,notgiven);
  socialsecuritynumber = 0..10000;
  rekord = record
    a : socialsecuritynumber;
    b : sex
  end;
var
  fyle : file of rekord;
begin
  rewrite(fyle);
  with fyle do
  begin
    a:=9999;
    b:=notgiven;
    put(fyle)
  end;
  writeln(' ERROR NOT DETECTED...6.6.5.2-7')
end.
```
program t6p6p5p3d1(output);
type
two = (a,b);
recone = record
  l : integer;
j : boolean
end;
rectwo = record
c : integer;
  case tagfield : two of
    a : (m : integer);
    b : (n : boolean)
  end;
recthree = record
  ptrone
  ptrtwo
  ptrthree
  c : integer;
  case tagfield : two of
    a : (case tagfield : two of
      a : (o : real);
      b : (p : char));
    b : (q : integer)
  end;
var
  ptrone : #recone;
  ptrtwo : #rectwo;
  ptrthree : #recthree;
begin
  new(ptrone);
  new(ptrtwo,a);
  ptrtwo1.tagfield:=a;
  new(ptrthree,a,b);
  ptrthree1.tagfield:=a;
  writeln(' PASS...6.6.5.3-1')
end.

program t6p6p5p3d2(output);
begin
  for i :=1 to 10 do
cbegin
    new(ptr);
    ptr:=i;
  end;
  writeln(' PASS...6.6.5.3-2')
end.

program t5p6p5p3d3(output);
type
  rekord = record
    a : integer;
    b : boolean
  end;
var
  ptr : #rekord;
begin
  ptr:=nil;
dispose(ptr);
  writeln(' ERROR NOT DETECTED...6.6.5.3-3')
end.
Similarly to 6.6.5.3-3, an error is caused by the pointer variable of dispose being used.
The error should be detected by the compiler or at run-time.

```pascal
type
  rekord = record
    a : integer;
    b : boolean
  end;
var
  ptr : *rekord;
begin
  dispose(ptr);
  writeln(' ERROR NOT DETECTED...6.6.5.3-5')
end.
```

This program causes an error to occur as a variable which is an element of the record-variable-list of a with statement is referred to by the pointer parameter of dispose.

```pascal
program t6p6p5p3d6(output);
type
  subrange = 0..9999;
  rekord = record
    name : packed array[1..15] of char;
    employeno : subrange
  end;
var
  ptr : *rekord;
begin
  with ptr do
  begin
    name := 'HARRY M. MULLER';
    employeno := 9998;
    dispose(ptr)
  end;
  writeln(' ERROR NOT DETECTED...6.6.5.3-6')
end.
```

This program causes an error to occur as a variable which is created by the use of the variant form of new is used as an operand in an expression.
The error should be detected by the compiler, or at run-time.

```pascal
program t5p6p5p3d7(output);
type
  two = (a,b);
  rekord = record
    case tagfield:two of
      a : (m : boolean);
      b : (n : char)
    end;
  end;
var
  ptr : *rekord;
  r : rekord;
begin
  new(ptr,a);
  ptr1 := true;
  r := ptr1;
  writeln(' ERROR NOT DETECTED...6.5.5.3-7')
end.
```
This test is similar to 6.6.5.3-7, except that the variable created is used as the variable in an assignment statement. The error should be detected by the compiler or at run-time.

```
program t6p6p3d9(output);
type
two = (a,b);
rekord = record
case tagfield:two of
  a : (m : boolean);
  b : (n : char)
end;
begin
  new(ptr,b);
  r.tagfield :=b;
  r.m:=true;
  writeln(' ERROR NOT DETECTED ... 6.6.5.3-8')
end.
```

This test is similar to 6.6.5.3-9, except that the variable created is used as an actual parameter. The error should be detected by the compiler or at run-time.

```
program t6p6p3d9(output);
type
two = (a,b);
rekord = record
case tagfield:two of
  a : (m : boolean);
  b : (n : char)
end;
var
  ptr : trekord;
begin
  new(ptr,a);
  writeln(' ERROR NOT DETECTED ... 6.6.5.3-9')
end.
```

This program tests that pack and unpack are implemented in this compiler as according to the Standard. The compiler fails if the program does not compile.

```
program t6p6p5d4d(output);
type
  colourtype = (red,pink,orange,yellow,green,blue);
var
  unone : array[3..24] of char;
  pacone : packed array[1..4] of char;
  untwo : array[4..8] of colourtype;
  pactwo : packed array[5..7] of colourtype;
  i : integer;
  colour : colourtype;
begin
  pacone:=' ABCD';
  unpack(pacone,unone,5);
  colour :=red;
  for i:=4 to A do
    begin
      untwo[i]:=colour;
      colour:=succ(colour)
    end;
  pack(untwo,5, pactwo);
  if unone[5]='A' then
    writeln(' PASS ... 6.6.5.4-1')
  else
    writeln(' FAIL ... 6.6.5.4-1')
end.
```
The Pascal Standard does not state what action takes place when a standard function is used as a functional parameter. The effect is implementation dependent. This program uses a standard function as a parameter to a procedure. The compiler may reject this as an error, or may permit it as it should other functional parameters.

```pascal
program t6p6p6pld1(output);
procedure quidnunk(function a(b:integer):boolean);
var
  x:integer;
  y:boolean;
begin
  x:=5;
  y:=a(x);
  if x=1 then
    writeln('STANDARD FUNCTION PERMITTED AS PARAMETERS', '...6.6.6.1-1')
  else
    writeln('STANDARD FUNCTION NOT PERMITTED AS ', 'PARAMETERS...6.6.6.1-1')
end;
begin
  quidnunk(odd)
end.
```

This program tests the implementation of the arithmetic function abs. Both real and integer expressions are used. The compiler fails if the program does not compile and run.

```pascal
program t6p6p6p2dl(output);
const
  pi = 3.1415926;
var
  i, counter:integer;
  r:real;
function myabs1(i:integer):integer;
begin
  if i<0 then
    myabs1:=i
  else
    myabs1:=i
end;
function myabs2(r:real):real;
begin
  if r<0 then
    myabs2:=r
  else
    myabs2:=r
end;
begin
  counter:=0;
  for i:=-10 to 10 do
    begin
      if abs(i)=myabs1(i) then
        counter:=counter+1
    end;
  r:=-10.3;
  while r<10.3 do
    begin
      if abs(r)=myabs2(r) then
        counter:=counter+1;
      r:=r+0.9
    end;
  if counter=44 then
    writeln('PASS...6.6.6.2-1:ABS')
  else
    writeln('FAIL...6.6.6.2-1:ABS')
end.
```
This program tests the implementation of the arithmetic function \( \text{sqr} \). Both real and integer expressions are used.

The compiler fails if the program does not compile and run.

program t6p6p6p2d2(output);
var
  i, counter: integer;
  variable: real;
begin
  counter := 0;
  for i := -10 to 10 do
    begin
      if \( \text{sqr}(i) = i \times i \) then
        counter := counter + 1;
    end;
  variable := -10.3;
  while (variable < 10.3) do
    begin
      if (\( \text{sqr}(\text{variable}) = \text{variable} \times \text{variable} \)) then
        counter := counter + 1;
      variable := variable + 0.9;
    end;
  if (counter = 44) then
    writeln(' PASS ...6.6.6.2-2')
  else
    writeln(' FAIL ...6.6.6.2-2: SQR');
end.

Program tests the implementation of the arithmetic functions \( \sin \), \( \cos \), \( \exp \), \( \ln \), \( \text{sqr} \), and \( \text{arctan} \).

A rough accuracy test is done, but is not the purpose of this program.

The compiler fails if the program does not compile and run.

program t6p6p6p2d3(output);
const
  pi = 3.1415926;
var
  counter: integer;
begin
  counter := 0;
  if (\( \sin(\pi) < 0.00001 \)) and
      ((0.70710 < \( \sin(\pi/4) \)) and
       (\( \sin(\pi/4) < 0.70711 \)) then
    counter := counter + 1
  else
    writeln(' FAIL ...6.6.6.2-3: SIN');

  if (\( \cos(\pi) < -0.99999 \)) and
      ((0.70710 < \( \cos(\pi/4) \)) and
       (\( \cos(\pi/4) < 0.70711 \)) then
    counter := counter + 1
  else
    writeln(' FAIL ...6.6.6.2-3: COS');

  if ((2.71828 < \( \exp(1) \)) and
      (\( \exp(1) < 2.71829 \)) and
      (\( (0.35787 < \exp(-1) \)) and
       (\( \exp(-1) < 0.36788 \)) and
       (\( 8103.08392 < \exp(9) \)) and
        (\( \exp(9) < 8103.08393 \)) then
    counter := counter + 1
  else
    writeln(' FAIL ...6.6.6.2-3: EXP');

  if ((ln(\( \exp(1) \)) > 0.99999) and
      ((0.69314 < ln(2)) and
       (\( \ln(2) < 0.69315 \)) then
    counter := counter + 1
  else
    writeln(' FAIL ...6.6.6.2-3: LN');

  if (\( \sqrt{25} = 5 \)) and
      ((5.09901 < \( \sqrt{26} \)) and
       (\( \sqrt{26} < 5.09902 \)) then
    counter := counter + 1
  else
    writeln(' FAIL ...6.6.6.2-3: SQRT');

  if ((0.09956 < arctan(0.1)) and
      (arctan(0.1) < 0.09967) and
      (arctan(0) = 0) then
    counter := counter + 1
  else
    writeln(' FAIL ...6.6.6.2-3: ARCTAN');

  if counter > 6 then
    writeln(' PASS ...6.6.6.2-3')
end.
This program causes an error to occur as an expression with a negative value is used as an argument for the arithmetic function ln.

The error should be detected at run-time.

program t6p6p6p2d4(output);
var
  m : real;
begin
  m := -2.71828;
  m := ln(m*2);
  writeln(' ERROR NOT DETECTED...6.6.6.2-4');
end.

This program causes an error to occur as a negative argument is used for the sqrt function.
The error should be detected at run-time.

program t6p6p6p2d5(output);
var
  m : real;
  i, j : integer;
begin
  i := 256;
  j := i*2;
  j := j - 257;
  m := sqrt(j - i);
  writeln(' ERROR NOT DETECTED...6.6.6.2-5');
end.

This test checks the implementation of the sqrt function.

program t6p6p6p2d6(output);
var

{ This subroutine is intended to determine the characteristics of the floating-point arithmetic system that are specified below. The first three are determined according to an algorithm due to M. Malcolm, CACM 15 (1972), pp. 949-951, incorporating some, but not all, of the improvements suggested by M. Gentleman and S. Marovich, CACM 17 (1974), pp. 276-277. The version given here is for single precision. }
ibeta is the radix of the floating-point representation
it is the number of base ibeta digits in the floating-point
significant
irnd = 0 if the arithmetic chops,
1 if the arithmetic rounds
ngrd = 0 if irnd=1, or if irnd=0 and only it base ibeta
digits participate in the post normalization shift
of the floating-point significant in multiplication
1 if irnd=0 and more than it base ibeta digits
participate in the post normalization shift of the
floating-point significant in multiplication
macheap is the exponent on the smallest positive floating-point
number eps such that 1.0+eps = 1.0
negeps is the exponent on the smallest positive fl. pt. no.
egeps such that 1.0-negeps = 1.0, except that
neeps is bounded below by -it-3
iexp is the number of bits (decimal places if ibeta=10)
reserved for the representation of a floating-point number
minexp is the exponent of the smallest positive fl. pt. no.
xmin = 1.0 minimum
maxexp is the exponent of the largest finite floating-point
number xmax
eps is the smallest positive floating-point number such
that 1.0-eps = 1.0. in particular,
eps = ibeta*macheap
epsneg is the smallest positive floating-point number such
that 1.0-eps = 1.0 (except that the exponent
neeps is bounded below by 11-3). in particular
epsneg = negeps
xmin is the smallest positive floating-point number. in
particular, xmin = ibeta ** minexp
xmax is the largest finite floating-point number, in
particular xmax = (1.0-epsneg) * ibeta ** maxexp
note - on some machines xmax will be only the
second, or perhaps third, largest number, being
too small by 1 or 2 units in the last digit of the
significant.

begin
irnd := 1;
one := ( irnd );
a := one + one;
b := a;
end;;

zero := 0.0;
{ determine ibeta,beta ala Malcolm
while (( ( a + one ) - a ) - one = zero ) do begin
a := a + a);
end;
while (( a + b ) - a = zero ) do begin
b := b + b;
end;
ibeta := trunc (( a + b ) - a);
beta := ( ibeta );
betan := beta - one;
{ determine irnd,ngrd,it
if (( a + betaml ) - a = zero ) then irnd := 0;
it := 0;
a := one;
repeat begin
it := it + 1;
a := a * beta;
end until (( ( a + one ) - a ) - one <> zero )
{ determine negep, epsneg
negep := it + 3;
a := one;
for i := 1 to negep do begin
a := a / beta;
end;
while (( ( one - a ) - one = zero ) do begin
a := a * beta;
negep := negep - 1;
end;
gep := negep - 1;
epsneg := a;
{ determine macheap, eps
macheap := negep;
while (( ( one + a ) - one = zero ) do begin
a := a * beta;
macheap := macheap + 1;
end;
eps := a;
{ determine ngrd
ngrd := 0;
if ((( irnd = 0 ) and (( one + eps ) * one - one ) <> zero )
thend
ngrd := 1;
{ determine iexp, minexp, xmin
loop to determine largest i such that
(1/beta) ** (2**(i))
does not underflow
exit from loop is signall by an underflow

i, iz, j, k, m, mx : integer;
a, b, beta, betain, betam1, one, y, z, zero : real;
underflo : boolean;

begin
irnd := 1;
one := ( irnd );
a := one + one;
b := a;
end;;


```pascal
1 := 0;
betain := one / beta;
z := betain;
underfl := false;
repeat begin
  y := z;
  n := y * y;
  if (( z * one = zero ) or ( abs ( z ) > y ) ) then begin
    underfl := true;
    i := 1 + 1;
    end;
  until underfl;
  k := 1;
  \{ determine k such that (1/beta)^k does not underflow \}
  first set k = 2 ** i
  for j := 1 to i do begin
    \{ for decimal machines only \}
    iexp := i + 1;
    mx := k + k;
    if ( ibeta = 10 ) then begin
      iz := ibeta;
      while ( k > iz ) do begin
        iz := iz * ibeta;
        if ( y > 2 ) then begin
          iexp := iexp + 1;
        end;
      end;
      mx := iz + iz - 1;
    end;
    underfl := false;
    repeat begin
      \{ loop to construct xmin \}
      exit from loop if signed by an underflow
      xmin := y;
      y := y * betain;
      if (( y * one = zero ) or ( abs ( y ) > xmin ) ) then begin
        xmin := y;
        end;
      end until underflo;
      \{ determine maxexp, xmin \}
      if ( ( mx <= k + k - 3 ) and ( ibeta <= 10 ) ) then begin
        maxexp := mx + minexp;
        xmin := xmin * beta;
      end;
      i := maxexp + minexp;
      if ( ( ibeta > 2 ) and ( i > 0 ) ) then begin
        maxexp := maxexp - 1;
        if ( i > 20 ) then maxexp := maxexp - 3;
        xmin := one - epsneg;
        \{ The quality of the random numbers is not important. \}
        if recoding is needed for small wordlength computers, even returning a constant value or zero is possible.\}
        \{ The value iy is global, and is initialized in the driver \}
        begin
          ly := ( iy*125 ) mod 2794200;
          random := ( iy ) / 2794200.000 ;
        end;
        if mx <= k + k - 3 do begin
          maxexp := k + k;
          xmin := x * beta;
        end;
        function randl ( x : real ) : real ;
        \{ returns pseudo random numbers logarithmically distributed over (1,exp(x)), thus \}
        \{ x randl(ln(b/a)) is logarithmically distributed in (ab), \}
        \{ other subroutines required \}
      exp(x) - the exponential routine
      random - a function program returning random real
      numbers uniformly distributed over (0,1). \}
```

The text above is a PASCAL program snippet, likely related to generating random numbers and managing underflows in a mathematical or scientific context. It includes comments and mathematical expressions that are typical of algorithms for handling floating-point arithmetic and random number generation.
begin
rand1 := exp ( x * random );
end;

procedure printtestrun ( n:integer; lb,ub:real; 
big, small : integer;
mean, maxerror, xerror, rmseerror : real); 
begin
writeln('5,n:4,' RANDOM ARGUMENTS WERE TESTED FROM THE INTERVAL.');
if (n <> 0.0) then begin
writeln('5,THE RESULT WAS TOO LARGE',big,5,' TIMES, AND');
end;
writeln('5,THE RESULT WAS TOO SMALL',small,5,' TIMES');
end;

begin 
if (mean < 0.0) then begin
writeln('5,MEAN=',mean:15,'=',' thoughtful of the arguments');
end;
 writeln('5,RELATIVE ERROR=',mean:15,'=',' thoughtful of the arguments');
end;

if (maxerror <> 0.0) then begin
writeln('5,THE MAXIMUM RELATIVE ERROR OF',maxerror:15, '=', ' thoughtful of the arguments');
end;
 writeln('5,THE MAXIMUM RELATIVE ERROR OF',maxerror:15, '=', ' thoughtful of the arguments');
 end;

if (rmseerror <> 0.0) then begin
writeln('5,ROOT-MEAN-SQUARE RELATIVE ERROR=',rmseerror:15, ' thoughtful of the arguments');
end;
 writeln('5,ROOT-MEAN-SQUARE RELATIVE ERROR=',rmseerror:15, ' thoughtful of the arguments');
end;

begin 
[ OF PRINT TEST RUN ]

beginchar ( ibeta , it , lnd , nqrd , machep , negep , iexp , minexp , 
maxexp , eps , epneg , xmin , xmax );
beta := ( ibeta );
sqbeta := sqrt ( beta );
end;

alpha := ln ( ibeta );
a := 1.0 / sqbeta ;
b := 1.0 ;
n := 2000 ;
y := 100000 .

random argument accuracy tests
begin
for j := 1 to 2 do begin
for i := 1 to n do begin
x := a * rand1 ( c );
y := x * x ;
z := sqrt ( y );
w := ( z - x ) / x ;
end;
if ( w > 0.0 ) then k := k + 1 ;
end;
end;

end. 
procedure machar (var ibeta, it, irnd, ngrd, machep, minexp, maxexp : integer var eps, epsneg, xmin, xmax : real);

This subroutine is intended to determine the characteristics
of the floating-point arithmetic system that are specified
by the standard subprograms required from this package.

subprograms required from this package
machar - as for sqrttest
random - as for sqrttest

abs, ln, arctan, sqrt

eps, epsneg, expon, iexp, iexp, k, kl, machep

eps is the smallest positive floating-point number such
that 1.0+eps > 1.0. In particular, eps = ibeta^machep

epsneg is the smallest positive floating-point number such
that 0 < epsneg < 1.0 (except that the exponent
negeps is bounded below by it-3). In particular
epsneg = ibeta^negep

xmin is the smallest positive floating-point number in
particular, xmin = ibeta^-negeps

This subroutine is intended to determine the characteristics
of the floating-point system that are specified
by the standard subprograms required from this package.

subprograms required from this package
machar - as for sqrttest
random - as for sqrttest

abs, ln, arctan, sqrt

eps, epsneg, expon, iexp, iexp, k, kl, machep

eps is the smallest positive floating-point number such
that 1.0+eps > 1.0. In particular, eps = ibeta^machep

epsneg is the smallest positive floating-point number such
that 0 < epsneg < 1.0 (except that the exponent
negeps is bounded below by it-3). In particular
epsneg = ibeta^negep

xmin is the smallest positive floating-point number in
particular, xmin = ibeta^-negeps

This subroutine is intended to determine the characteristics
of the floating-point system that are specified
by the standard subprograms required from this package.

subprograms required from this package
machar - as for sqrttest
random - as for sqrttest

abs, ln, arctan, sqrt

eps, epsneg, expon, iexp, iexp, k, kl, machep

eps is the smallest positive floating-point number such
that 1.0+eps > 1.0. In particular, eps = ibeta^machep

epsneg is the smallest positive floating-point number such
that 0 < epsneg < 1.0 (except that the exponent
negeps is bounded below by it-3). In particular
epsneg = ibeta^negep

xmin is the smallest positive floating-point number in
particular, xmin = ibeta^-negeps

This subroutine is intended to determine the characteristics
of the floating-point system that are specified
by the standard subprograms required from this package.

subprograms required from this package
machar - as for sqrttest
random - as for sqrttest

abs, ln, arctan, sqrt

eps, epsneg, expon, iexp, iexp, k, kl, machep

eps is the smallest positive floating-point number such
that 1.0+eps > 1.0. In particular, eps = ibeta^machep

epsneg is the smallest positive floating-point number such
that 0 < epsneg < 1.0 (except that the exponent
negeps is bounded below by it-3). In particular
epsneg = ibeta^negep

xmin is the smallest positive floating-point number in
particular, xmin = ibeta^-negeps

This subroutine is intended to determine the characteristics
of the floating-point system that are specified
by the standard subprograms required from this package.

subprograms required from this package
machar - as for sqrttest
random - as for sqrttest

abs, ln, arctan, sqrt

eps, epsneg, expon, iexp, iexp, k, kl, machep

eps is the smallest positive floating-point number such
that 1.0+eps > 1.0. In particular, eps = ibeta^machep

epsneg is the smallest positive floating-point number such
that 0 < epsneg < 1.0 (except that the exponent
negeps is bounded below by it-3). In particular
epsneg = ibeta^negep

xmin is the smallest positive floating-point number in
particular, xmin = ibeta^-negeps

This subroutine is intended to determine the characteristics
of the floating-point system that are specified
by the standard subprograms required from this package.

subprograms required from this package
machar - as for sqrttest
random - as for sqrttest

abs, ln, arctan, sqrt

eps, epsneg, expon, iexp, iexp, k, kl, machep

eps is the smallest positive floating-point number such
that 1.0+eps > 1.0. In particular, eps = ibeta^machep

epsneg is the smallest positive floating-point number such
that 0 < epsneg < 1.0 (except that the exponent
negeps is bounded below by it-3). In particular
epsneg = ibeta^negep

xmin is the smallest positive floating-point number in
particular, xmin = ibeta^-negeps

This subroutine is intended to determine the characteristics
of the floating-point system that are specified
by the standard subprograms required from this package.

subprograms required from this package
machar - as for sqrttest
random - as for sqrttest

abs, ln, arctan, sqrt

eps, epsneg, expon, iexp, iexp, k, kl, machep

eps is the smallest positive floating-point number such
that 1.0+eps > 1.0. In particular, eps = ibeta^machep

epsneg is the smallest positive floating-point number such
that 0 < epsneg < 1.0 (except that the exponent
negeps is bounded below by it-3). In particular
epsneg = ibeta^negep

xmin is the smallest positive floating-point number in
particular, xmin = ibeta^-negeps
for i := 1 to ngrd do begin
    iexp := i + 1;
    mx := k + 1;
    if ( ibeta = 10 ) then begin
        { for decimal machines only
            iexp := 2; iz := ibeta; 
            while ( ( k >= iz ) do begin
                iz := iz * ibeta; iexp := iexp + 1; 
            end; 
            mx := iz + 1; 
            end; 
            underflo := false; repeat begin
                { loop to construct xmin 
                    exit from loop is signalled by an underflow
                    xmin := y; y := y * betain; 
                    if ( ( y * one ) = zero ) or ( abs ( y ) >= xmin )
                        then begin
                            underflo := true; 
                            end else begin
                                k := k + 1; 
                                end; 
                            end until underflo; 
                end; 
            end; 
        } 
        determine maxexp, xmax 
        if ( ( mx <= k + k - 3 ) and ( ibeta <> 10 ) ) then begin
            maxexp := mx + maxexp; 
            maxexp := mx * one + minexp; 
            { adjust for machines with implicit leading
                bit in binary significand and machines with
                radix point at extreme right of significand
                if ( ( iexp = 10 ) and ( i = 0 ) ) then maxexp := maxexp - 1;
                if ( ( iexp = 10 ) and ( i = 0 ) ) then maxexp := maxexp - 3; 
                if ( ( iexp = 10 ) and ( i = 0 ) ) then maxexp := one - openep; 
                xmax := ( openep * betain * betain + betain ) / xmin; 
            end; 
            maxexp := maxexp + 1; 
            if ( ( iexp = 10 ) and ( i = 0 ) ) then begin
                for j := 1 to i do begin
                    xmax := xmax * beta; 
                    end; 
            end; 
        end; 
    } 
end; 
{ determine k such that (l/beta)**k does not underflow
for j := 1 to i do begin
    k := k + k; 
end; 
if ( ibeta = 10 ) then begin
    { for decimal machines only
        iexp := 2; iz := ibeta; 
        while ( ( k >= iz ) do begin
            iz := iz * ibeta; iexp := iexp + 1; 
        end; 
        mx := iz + 1; 
        end; 
        underflo := false; repeat begin
            { loop to construct xmin 
                exit from loop is signalled by an underflow
                xmin := y; y := y * betain; 
                if ( ( y * one ) = zero ) or ( abs ( y ) >= xmin )
                    then begin
                        underflo := true; 
                        end else begin
                            k := k + 1; 
                            end; 
                        end until underflo; 
                end; 
            end; 
        } 
        determine maxexp, xmax 
        if ( ( mx <= k + k - 3 ) and ( ibeta <> 10 ) ) then begin
            maxexp := mx + maxexp; 
            maxexp := mx * one + minexp; 
            { adjust for machines with implicit leading
                bit in binary significand and machines with
                radix point at extreme right of significand
                if ( ( iexp = 10 ) and ( i = 0 ) ) then maxexp := maxexp - 1;
                if ( ( iexp = 10 ) and ( i = 0 ) ) then maxexp := maxexp - 3; 
                if ( ( iexp = 10 ) and ( i = 0 ) ) then maxexp := one - openep; 
                xmax := ( openep * betain * betain + betain ) / xmin; 
            end; 
            maxexp := maxexp + 1; 
            if ( ( iexp = 10 ) and ( i = 0 ) ) then begin
                for j := 1 to i do begin
                    xmax := xmax * beta; 
                    end; 
            end; 
        end; 
    } 
end;
begin

function random : real ;
{
 random number generator - based on algorithm 266
 by Pike and Hill (modified by Hanson)
 collected Alg. from CACM.

This subprogram is intended for use on computers with
fixed point wordlength of at least 29 bits. It is
best if the floating point significand has at most
29 bits. }

The quality of the random numbers is not important.
If recoding is needed for small wordlength computers,
even returning a constant value or zero is possible. }

The value iy is global, and is initialized in the driver }
begin

ly := (ly*125) mod 2796203;
random := (iy)/2796203.0e0 ;
end;

procedure printtestrun (n:integer; lb,ub:real);
big,small : integer;
mean,maxerror,rmse rror:real);
beg in
writeln (':5,n:4,', 'RANDOM ARGUMENTS WERE TESTED FROM THE INTERVAL');
writeln (':10, ('',lb:15,'',',',ub:15,''));
writeln (':10, 'THE RESULT WAS TOO LARGE,big:5,' TIMES, AND');
writeln (':10, 'TOO SMALL,small:5,' TIMES');
 writeln (':10, 'MEAN RELATIVE ERROR=',mean:15,'=',
 'IBETA:4,' ** ,LN (ABS (mean))/ALB ETA:7:2);
 writeln (':10, 'MAXIMUM RELATIVE ERROR=',maxerror:15,'=',
 'IBETA:4,' ** ,LN (ABS (maxerror))/ALB ETA:7:2);
 writeln (':10, 'ROOT-MEAN-SQUARE RELATIVE ERROR=',rmse rror:15,
 'IBETA:4,' ** ,LN (ABS (rmse rror))/ALB ETA:7:2);
 writeln (': OF PRINT TEST RUN }}
begin

ly := 100001;
 machar ( ibeta , it , irnd , ngrd , machep , negep , loexp , min exp ,
 max exp , eps , epemax , xmin , xmax );
beta := ( ibeta );
alpha := ln ( beta );

one := 1.0 ;
half := 0.5 ;
zero := 0.0 ;
a := - 0.0625 ;
b := - a ;
ob32 := b * half ;
n := 2000 ;
xn := ( n );
l1 := 0 ;

for j := 1 to 4 do begin

end;

00571400
00574200
00574300
00574400
00574500
00574600
00574900
00575000
00575100
00575200
00575300
00575400
00575500
00575600
00575700
00575800
00575900
00576000
00576100
00576200
00576300
00576400
00576500
00576600
00576700
00576800
00576900
00577000
00577100
00577200
00577300
00577400
00577500
00577600
00577700
00577800
00577900
00578000
00578100
00578200
00578300
00578400
00578500
00578600
00578700
00578800
00578900
00579000
00579100
00579200
00579300
00579400
00579500
00579600
00579700
00579800
00579900
00580000
00580100
00580200
00580300
00580400
00580500
00580600
00580700
00580800
00580900
00581000
00581100
00581200
00581300
00581400
00581500
00581600
00581700
00581800
00581900
00582000
00582100
00582200
00582300
00582400
00582500
00582600
00582700
00582800
00582900
00583000
00583100
00583200
00583300
00583400
00583500
00583600
00583700
00583800
00583900
00584000
00584100
00584200
00584300
00584400
00584500
00584600
00584700
00584800
00584900
00585000
00585100
00585200
00585300
00585400
00585500
00585600
00585700
00585800
00585900
00586000
00586100
00586200
if ( w > r6 ) then begin
    r6 := w;
    xl := x;
end;
r7 := r7 + w * w;
xl := xl + del;
end;

r5 := r5 / xn;
r7 := sqrt ( r7 / xn );
if ( j = 1 ) then begin
    writeln(' TEST OF ARCTAN(X) VS TRUNCATED TAYLOR SERIES');
    writeln;
    end;
if ( j = 2 ) then begin
    writeln(' TEST OF ARCTAN(X) VS ARCTAN(1/16) + ');
    writeln(' ARCTAN(X-1/16)/(1+X/16)');
    writeln;
    end;
if ( j > 2 ) then begin
    writeln(' TEST OF 2*ARCTAN(X) VS ARCTAN(2X/(1-X*X)');
    writeln;
end;
printtestrun(n,a,b,k1,r6,r7,xl);

a := b;
if ( j = 1 ) then b := 2.0 - sqrt ( 3.0 );
if ( j = 2 ) then b := sqrt ( 2.0 ) - one;
if ( j > 2 ) then b := one;

special tests

writeln(' THE IDENTITY ARCTAN(-X) = -ARCTAN(X) WILL BE TESTED');
writeln(' ', 'X', ' ', 'F(X) + F(-X)');
writeln;
a := 5.0;

for i := 1 to 5 do begin
    x := random * a;
    z := arctan ( x ) + arctan ( - x );
    writeln(x:14, z:15);
end;
writeln;

writeln(' THE IDENTIT Y ARCTAN(X) = X, X SMALL, WILL BE TESTED');
writeln(' ', 'X', ' ', 'F(X) - F(X)');
writeln;

betap := exp ( it * ln( beta ));
x := random / betap;

for i := 1 to 5 do begin
    y := arctan ( x ) + arctan ( - x );
    writeln(' ', 'X', ' ', 'Y', ' ', 'Y-X');
end;
writeln;

x := exp ( expos * ln( beta ));
y := arctan ( x );
writeln(' ', 'X', ' ', 'Y', ' ', 'X-Y');
writeln;

y := arctan ( x / y );
writeln(' THE FUNCTION WILL BE CALLED WITH THE ARGUMENT');
writeln(x:14, y:13);
writeln;
z := arctan ( xmax );
writeln(' ', 'X', ' ', 'Z', ' ', 'X-Z');
writeln;
write(' THIS CONCLUDES THE TESTS');
end.
as for sqrt

program t6p6p6p2d8(output);

var

{ data required

none

other subprograms in this package

machar - as for sqrt

random - as for sqrt

standard subprograms required

abs, In, exp, sqrt

machep, irnd, it, irnd, ngrd, irnd; integer

a, alpha, b, beta, d, del, eps, epsneg, r5, r6, r7, v

w, x, xl, xmax, xmin, xn, xl, y, z, zz; real

procedure machar (var ibeta, it, irnd, ngrd, machep, ngrd: integer; var eps, epsneg, xmin, xmax: real); 0

var

{ This subroutine is intended to determine the characteristics

of the floating-point arithmetic system that are specified

below. The first three are determined according to an

algorithm due to M. Malcolm, CACM 15 (1972), pp. 949-951,

incorporating some, but not all, of the improvements

suggested by M. Gentleman and S. Marovich, CACM 17 (1974),

pp. 276-277. The version given here is for single precision.

Latest revision - October 1, 1976.

Author - W. J. Cody

Argonne National Laboratory

Revised for Pascal - R. A. Frek

University of Tasmania

Hobart

Tasmania

ibeta is the radix of the floating-point representation

it is the number of base ibeta digits in the floating-point

significant

irnd = 0 if the arithmetic chops,

1 if the arithmetic rounds

ngrd = 0 if irnd=1, or if irnd=0 and only it base ibeta

digits participate in the post normalization shift

of the floating-point significant in multiplication

1 if irnd=0 and more than it base ibeta digits

participate in the post normalization shift of the

floating-point significant in multiplication

machep is the exponent on the smallest positive floating-point

number eps such that 1.0+eps <> 1.0

negeps is the exponent on the smallest positive fl. pt. no.

such that 1.0-negeps <> 1.0, except that

negeps is bounded below by it-3. In particular

eps = ibeta**machep

epsneg = ibeta**negeps

xsn is the smallest positive floating-point number. In

particular, xsn = ibeta ** minexp

xmax is the largest finite floating-point number. In

particular, xmax = (1.0-epsneg) * ibeta ** maxexp

procedure machar (var ibeta, it, irnd, ngrd, machep, ngrd: integer; var eps, epsneg, xmin, xmax: real); 0

var

i, irn, irnd, it, j, k, kl, machep, 

mymax, irnd, it, irnd, ngrd, machep, negep, irnd, ngrd: integer;

a, alpha, b, beta, d, del, eps, epsneg, r5, r6, r7, v

w, x, xl, xmax, xmin, xn, xl, y, z, zz; real

procedure machar (var ibeta, it, irnd, ngrd, machep, negep, irnd, ngrd, maxexp, minexp: integer; var eps, epsneg, xmin, xmax: real); 0
repeat begin
  it := it + 1;
  a := a * beta;
end until (( (a + one) - a) - one <> zero);

{ determine negep, epsneg }
negep := it + 3;
a := one;
for i := 1 to negep do begin
  a := a / beta;
end;

while (( (one - a) - one = zero) do begin
  a := a * beta;
  negep := negep - 1;
end;
negep := - negep;
epsneg := a;

{ determine machepl, epspl }
machepl := negep /
while (( (one + a) - one = zero) do begin
  a := a * beta;
  machepl := machepl + 1;
end;

{ determine ngrd }
if i = 0) and(( (one + eps) * one = one) <> zero) then
  ngrd := 1;

{ determine iexp, minexp, xamin }
loop to determine largest i such that
  (1/beta) ** (2**i)) does not underflow
exit from loop is signalled by an underflow
i := 0;
betain := one / beta;
zedf := betain;
underflo := false;
repeat begin
  y := z;
  z := y * y;

{ check for underflow }
if ( (z * one = zero) or (abs (z > y)) then
  underflo := true;
end else begin
  k := 1 + 1;
end;
end until underflo;
k := 1;

{ determine k such that (1/beta)**k does not underflow }
first set k = 2 ** i

for j := 1 to i do begin
  k := k + k;
end;
iexp := i + 1;
mx := k + k;
if ( (ibeta = 10) ) then begin
  for decimal machines only
  lexp := 2;
  iz := ibeta;
  while ( k >= iz ) do begin
    iz := iz * ibeta;
    lexp := lexp + 1;
  end;
  mx := iz + iz - 1;
end;

underflo := false;
repeat begin
  loop to construct xamin
  exit from loop is signalled by an underflow
  xamin := y;
  y := y * betain;
  if ( (y = zero) or (abs (y) > xamin) ) then begin
    underflo := true;
  end else begin
    k := k + 1;
  end;
end until underflo;

minexp := - k;

{ determine maxexp, xamx }
if ( (imax <= k + k - 1) and (ibeta <> 10) ) then begin
  xamin := xamx + xamx;
end;

maxexp := xamx + minexp;

{ adjust for machines with implicit leading bit in binary significant and machines with
radix point at extreme right of significand }
if ( (ibeta = 2) and (i = 0) ) then maxexp := maxexp - 1;
if (i > 20) then maxexp := maxexp - 3;

xamx := one - epsneg;
if (xamx * one <> xamx) then xamx := one - beta * epsneg;

i := maxexp + minexp + 3;
if (i > 0) then begin
  for j := 1 to i do begin
    xamx := xamx * beta;
  end;
function random : real;

{ random number generator - based on algorithm 266
  by Pike and Hill (modified by Hanson)
collected Alg. from CACM.

This subprogram is intended for use on computers with
fixed point wordlength of at least 29 bits. It is
best if the floating point significand has at most
29 bits. }

{ The quality of the random numbers is not important.
If recoding is needed for small wordlength computers,
even returning a constant value or zero is possible. }
{ The value iy is global, and is initialized in the driver }

begin
  iy := (iy*125)+2796203;
  random := iy/2796203.0e0;
end;

procedure printtestrun( n:integer; lb,ub:real);
big/small : integer;
mean,maxerror,maxerrormean,rmserror:real;

begin
  writeln('15,n:4,' RANDOM ARGUMENTS WERE TESTED FROM THE INTERVAL:');
  writeln('10,0,lb:=15.,ub:=15.')
  writeln('5,THE RESULT WAS TOO LARGE,big:=5 TIMES, AND');
  writeln('10,TOO SMALL,small:=5 TIMES');
  if (mean <> 0.0) then begin
    writeln('5,MEAN RELATIVE ERROR =','mean:15.',=',
            IBETA:4, **',IN(ABS(mean))/ALPHA:7:2);
    mean2200;
  end;
  if (maxerror<> 0.0) then begin
    writeln('5,THE MAXIMUM RELATIVE ERROR OF','maxerror:15.',=',
            IBETA:4, **',IN(ABS(maxerror))/ALPHA:7:2);
    writeln('10,occurred for X =','maxerror:15');
    end;
  if (rmserror<> 0.0) then begin
    writeln('5,ROOT-MEAN-SQUARE RELATIVE ERROR =','rmserror:15.',=',
            IBETA:4, **',IN(ABS(rmserror))/ALPHA:7:2);
    end;
  writeln;
end; [ OF PRINT TEST RUN ]

begin
  iy := 100001;
  macchar ( ibeta , it , irnd , ngrd , machep , negep , iexp , minexp ,
            maxexp , eps , epsneg , xmin , xmax );
  beta := ( ibeta );
\begin{verbatim}
a := -10.0 * b;
b := 4.0 * xmin * \text{exp} \left( \text{it} \times \ln(\beta) \right);
end;
writeln(' THE IDENTITY EXP(X) * EXP(-X) - 1.0 WILL BE TESTED.'); writeln(' \text{x:13}, "y := exp ( x) - 1.0; writeln(x:15, z:15)\); end;

for i := 1 to 5 do begin
  x := random * \beta;
y := -x;
z := \text{exp} \left( x \right) * \text{exp} \left( y \right) - 1.0;
writeln(x:15, z:15);
end;
writeln(' TEST OF SPECIAL ARGUMENTS'); writeln;
x := 0.0;
y := \text{exp} \left( x \right) - 1.0;
writeln(' EXP(0,0) =', y:15);
writeln;
x := \text{trunc} \left( \ln \left( x_{\text{min}} \right) \right);
y := \text{exp} \left( x \right);
writeln(' \text{EXP}', x:13, ' =', z:15);
writeln;
x := \text{trunc} \left( \ln \left( x_{\text{max}} \right) \right);
y := \text{exp} \left( x \right);
writeln(' \text{EXP}', x:13, ' =', y:15);
writeln;
x := x / 2.0;
y := x / 2.0;
z := \text{exp} \left( v \right);
writeln(' IF EXP', x:13, ', IS NOT ABOUT'); writeln(' EXP', x:13, ' =', y:15);
writeln(' REDUCTION ERROR?'); writeln;

end.

{ TEST 5.6.6.2-9, CLASS=QUALITY }
{ This test checks the implementation of the sin and cos functions. }
program t566629(output);
var
  \text{data required}
  \text{none}
other subprograms in this package
machar - \text{as for sqrt test}
random - \text{as for sqrt test}

standard subprograms required
abs, ln, exp, cos, sin, sqrt

i, ibeta, iexp, irnd, it, il, j, k, kl, machep, ly, maxexp, minexp, n, negexp, ngrd: integer;
a, albeta, b, beta, betap, c, del, eps, epsneg, expn, r5, r6, r7, w, x, xl, xmax, xmin, xn, xl, y, z, zz: real;

procedure machar (var ibeta, it, irnd, ngrd, machep, negexp, iexp, minexp, maxexp: integer; var eps, epsneg, xmin, xmax: real);

{ This subroutine is intended to determine the characteristics of the floating-point arithmetic system that are specified below. The first three are determined according to an algorithm due to M. Gentleman and S. Marovich, CACM 17 (1974), pp. 275-277. The version given here is for single precision. }

{ TEST 6.6.3-1, CLASS=QUALITY }

\text{Author: W. J. Cody}
Argonne National Laboratory

\text{Revised for Pascal - R. A. Preck}
University of Tasmania
Hobart
Tasmania

ibeta is the radix of the floating-point representation
it is the number of base ibeta digits in the floating-point significand.
irnd = 0 if the arithmetic chops,
1 if the arithmetic rounds
ngrd = 0 if irnd=1, or if irnd=0 and only it base ibeta digits participate in the post normalization shift.

\text{PAGE 12}
\text{PASCAL NEWS #16}
\text{OCTOBER, 1979}
if irnd=0 and more than 1 base ibeta digits participate in the post normalization shift of the floating-point significand in multiplication

machep is the exponent on the smallest positive floating-point number such that 1.0+eps < 1.0

negeps is the exponent on the smallest positive fl. pt. no. negeps such that 1.0-negeps < 1.0, except that negeps is bounded below by 1e-3

eexp is the number of bits (decimal places if ibeta = 10) reserved for the representation of the exponent of a floating-point number

minexp is the exponent of the smallest positive fl. pt. no.

maxexp is the exponent of the largest finite floating-point number xmax

eps is the smallest positive floating-point number such that 1.0-eps < 1.0, in particular, eps = ibeta**machep

epsneg is the smallest positive floating-point number such that 1.0-eps < 1.0 (except that the exponent negeps is bounded below by 1e-3). in particular epsneg = ibeta**negeps

xmin is the smallest positive floating-point number. in particular, xmin = ibeta**minexp

xmax is the largest finite floating-point number. In particular, xmax = (1.0-epsneg) * ibeta ** maxexp note - on some machines xmax will be only the second, or perhaps third, largest number, being too small by 1 or 2 units in the last digit of the significand.

i , iz , j , k , mx : integer;
a , b , beta , betain , betaal , one , y , z , zero : real;
derfio : boolean;

begin
irnd := 1;
one := ( irnd );
a := one + one ;
b := a ;
zero := 0.0 ;
begin determine ibeta,beta ala Malcolm

while ( ( ( a + one ) - a ) - one = zero ) do begin
a := a + a ;
end ;
while ( ( a + b ) - a = zero ) do begin
b := b + b ;
end ;
ibeta := trunc ( ( a + b ) - a ) ;
beta := ( ibeta ) ;
betaal := beta - one ;
end
if ( ( ( a + betaal ) - a = zero ) then irnd := 0 ;
it := 0 ;
a := one ;
repeat begin

i := irnd + 1 ;
a := a * beta ;
end until ( ( ( a + one ) - a ) - one <> zero ) ;
{ determine negep , epsneg

negep := it + 3 ;
a := one ;
for i := 1 to negep do begin
a := a * beta ;
end ;
while ( ( one - a ) - one = zero ) do begin
a := a * beta ;
negep := negep - 1 ;
end ;
negep := - negep ;
end;
{ determine machep , eps

machep := negep ;
while ( ( one + a ) - one = zero ) do begin
a := a * beta ;
machep := machep + 1 ;
end ;
eps := a ;
end
{ determine ngrd

ngrd := 0 ;
if( ( irnd = 0 ) and( (( one + eps ) * one - one ) <> zero ) then

i := 0 ;
betain := one / beta ;
z := betain ;
derfio := false ;
repeat begin
y := z ;
z := y * y ;
end
{ check for underflow
if ( ( z * one = zero ) or ( abs ( z ) > y ) ) then begin
nderfio := true ;
end else begin
i := i + 1 ;
end ;
eneg until underfio ;
k := 1 ;
determine k such that \((l/beta)^k\) does not underflow

first set \(k = 2^* 1\)

for \(j = 1\) to \(i\) do begin

\(k := k + k\)

end;

\(iexp := i + 1\)

\(mx := k + k\)

if (ibeta <> 10) then begin

for decimal machines only

\(iexp := 2\)

\(iz := ibeta\)

while (k >= iz) do begin

\(iz := iz * ibeta\)

\(iexp := iexp + 1\)

end;

\(mx := iz + i - 1\)

end;

underflo := false;

repeat begin

loop to construct xmin

exit from loop if signal is underflow

\(xmin := y\)

\(y := y * betain\)

if (\((y < one)\) or (abs\(y\) > xmin)) then begin

then begin

underflo := true;

\(k := k + 1\)

end else begin

\(k := k + 1\)

end until underflo;

\(minexp := -k\)

{ determine maxexp, xmax }

if (\((mx <= k + k - 3)\) and (ibeta < 10)) then begin

\(mx := mx + mx\)

\(iexp := iexp + 1\)

end;

\(maxexp := mx + minexp\)

\{ adjust for machines with implicit leading

bit in binary significand and machines with

radix point at extreme right of significand

\(i := maxexp + minexp\)

\(i := maxexp + minexp - 1\)

\(i := maxexp - 1\)

\(i := maxexp - 3\)

\(xmax := one * epsneg\)

\(xmax := one - beta * epsneg\)

\(xmax := (xmas * betain) / xmin\)

\(i := maxexp + minexp + 3\)

if (\(i < 0\)) then begin

\(for \(j = 1\) to \(i\) do begin\)

\(xmas := xmax * beta\)

\(end;\)

end;

function random : real;

{ random number generator - based on algorithm 266

by Pike and Hill (modified by Hanson)

collected Alg. from CNOA.}

This subprogram is intended for use on computers with

fixed point wordlength of at least 29 bits. It is

best if the floating point significand has at most

29 bits.}

\{ The quality of the random numbers is not important.

If recoding is needed for small wordlength computers,

even returning a constant value or zero is possible. \}

\{ The value iy is global, and is initialized in the driver \}

begin

\(iy := (iy*125) \bmod 2796203\)

\(random := (iy)/2796203.0e0\)

end;

procedure printtestrun (ni:integer; lb,ub:real;

\(big,small: integer;\)

mean,maxerror,xmaxerror,rmerror:real);

begin

\(write(5,' RANDOM ARGUMENTS WERE TESTED FROM THE INTERVAL');\)

\(write(10,('','lb:15','','ub:15',','));\)

\(write(10,ibeta:4,' ** ',ln(abs(maxerror))/albota:7:2);\)

\(write(10,' THE RESULT WAS TOO LARGE',ibeta:4,' TIMES, AND');\)

\(write(10,' TOO SMALL',ibeta:4,' TIMES');\)

\(if (mean < 0,0) then begin\)

\(write(10,' THE MAXIMUM RELATIVE ERROR IS',maxerror:15,'\',ibeta:4,'** LN(abs(mean))/ALBOTA:7:2);\)

\(write(10,' THE MAXIMUM RELATIVE ERROR FOR X =',maxerror:15,end;\)

\(if (mserror <> 0,0) then begin\)

\(write(10,' THE NOMINAL SQUARE RELATIVE ERROR IS',rmerror:15,'\',ibeta:4,'** LN(abs(maxerror))/ALBOTA:7:2);\)

\(write(10,' THE NOMINAL SQUARE RELATIVE ERROR FOR X =',rmerror:15,end;\)

\{ OF PRINT TEST RUN \}

begin

\(iy := 9000000\)

\(mach (ibeta, it, irnd, ngrd, machep, negep, iexp, minexp,\)

\(machep, eps, epneg, xmin, xmax);\)

\(beta := (ibeta);\)
special tests

c := 1.0 / exp ( ( it div 2 ) * ln( beta ));
z := ( sin ( a + c ) - sin ( a - c ) ) / ( c + c );
write(' IF ', z;15, ' IS NOT ALMOST 1.0 THEN SIN HAS THE WRONG ');
writeint('PERIOD');
write;
writeint('THE IDENTITY -SIN(X) = -SIN(X) WILL BE TESTED');
writeint(':7, 'X', :9,'F(X) + F(-X)');
writeint;
writeint;for i := 1 to 5 do begin
x := random * a ;
z := sin ( x ) * sin ( - x );
writeint(x;14,z;15);
end ;
writeint;
writeint('THE IDENTITY SIN(X) = X, X SMALL, WILL BE TESTED.');
writeint(':7, 'X', :9,'F(X)');
writeint;
writeint;for i := 1 to 5 do begin
x := random * a ;
writeint(x;14,z;15);
end ;
writeint;
writeint('THE IDENTITY COS(-X) = COS(X) WILL BE TESTED.');
writeint(':7, 'X', :9,'F(X) - F(-X)');
writeint;
betap := exp ( it * ln( beta ));
x := random / betap ;
writeint;x
writeint;for i := 1 to 5 do begin
z := x - sin ( x ) ;
writeint(x;14,z;15);
end ;
writeint;
writeint('THE IDENTITY COS(X) = COS(X) WILL BE TESTED.');
writeint(':7, 'X', :9,'F(X)');
writeint;
writeint;for i := 1 to 5 do begin
x := random * a ;
z := cos ( x ) - cos ( - x );
writeint(x;14,z;15);
end ;
writeint;
writeint('THE FOLLOWING THREE LINES ILLUSTRATE THE LOSS IN');
writeint('SIGNIFICANCE FOR LARGE ARGUMENTS. THE ARGUMENTS');
writeint('USED ARE CONSECUTIVE.');
writeint;
writeint;z := sqrt ( betap );
x := x * ( 1.0 - explog ) ;
y := sin ( x );
writeint(':5, 'SIN(', x;15, ') = ', y;15);
writeint;
writeint;
writeln;
\[ x := z \times (1.0 + \varepsilon) \];
\[ y := \sin(x) \];
writeln(' \text{SIN(} x \text{)} y:15);" 

writeln;
\[ x := \text{beta} \];
writeln(' \text{SIN}(X) \text{ WILL BE CALLED WITH THE ARGUMENT } y = \sin(x) \];
writeln(' \text{SIN} \text{RETURNED THE VALUE } y:15);" 

\text{THIS CONCLUDES THE TESTS.};"

end.

---

\text{program t5p6p6p2d10(output);}
\begin{verbatim}
var
  data required

  none

  other subprograms in this package

  machar - as for sqrtest

  ran(k) - as for sqrtest

  standard subprograms required

  abs, ln, sqrt

i, ibeta, il, irnd, it, j, k, kl, macheap, iy, maxexp, minexp, n, negep, ngrd : integer;
a, albeta, b, beta, d, del, eight, eps, epsneg, half, r5, r7, r7, tenth, w, x, x1, xam, xam, xn, x1, y, z, z,
real;
\end{verbatim}

\text{procedure machar (var ibeta, it, irnd, ngrd, macheap, negep, lexp, minexp, maxexp : integer ; var eps, epsneg, xam, xam : real ;)

var

\{ This subroutine is intended to determine the characteristics of the floating-point arithmetic system that are specified below. The first three are determined according to an algorithm due to M. Malcolm, CACM 15 (1972), pp. 949-951, incorporating some, but not all, of the improvements suggested by M. Gentleman and S. Marovich, CACM 17 (1974), pp. 276-277. The version given here is for single precision.\}

\text{Latest revision - October 1, 1976.}

Author - W. J. Cody
Argonne National Laboratory

Revised for Pascal - R. A. Preak
University of Tasmania
Hobart
Tasmania

ibeta \text{ is the radix of the floating-point representation}

it \text{ is the number of base ibeta digits in the floating-point significand}

irnd \text{ = 0 if the arithmetic chops, 1 if the arithmetic rounds}

ngrd \text{ = 0 if irnd=1, or if irnd=0 and only it base ibeta digits participate in the post normalization shift}
of the floating-point significand in multiplication
define the exponent on the smallest positive floating-point number eps such that 1.0eps <> 1.0

negeps is the exponent such that 1.0-negeps <> 1.0, except that

negeps is bounded below by 1=3

iexp is the number of bits (decimal places) if beta = 10

reserved for the representation of the exponent of a floating-point number

minexp is the exponent of the smallest positive fl. pt. no.

mxexp is the exponent of the largest finite floating-point number

eps is the smallest positive floating-point number such that 1.0+eps <> 1.0, in particular,

eps = ibeta**machep

epsneg is the smallest positive floating-point number such that 1.0-eps <> 1.0 (except that the exponent

negeps is bounded below by its3). in particular

epsneg = ibeta**negep

xmin is the smallest positive floating-point number. in

particular, xmin = ibeta ** minexp

xmax is the largest finite floating-point number. in particular

xmax = (1.0-epsneg) * ibeta ** maxexp

note - on some machines xmax will be only the second, or perhaps third, largest number, being

too small by 1 or 2 units in the last digit of the significand.

begin

irnd := 1;
a := one + a;
while ((a + one) - a - one) <> zero do begin

end;

a := one;
repeat begin

i := i + 1;
a := a * ibeta ngrd, irnd := ibeta,beta ala Malcolm

while ((a + one) - a - one = zero) do begin

end;

a := one / ibeta;
function random : real;

begin
  y := (iy*125) mod 2796203;
  random := (y)/2796203.0e0;
end;
}

procedure printtestrun (n:integer; lb, ub:real;
big,small : integer;
mean,maxerror,rmaxerror,rmserror:real);
begin
  writeln(' :5:n:4,’ RANDOM ARGUMENTS WERE TESTED FROM THE INTERVAL’);
  writeln(' :10,’(lb,ub):15,’(lb,ub):15,’);
  writeln;
  writeln(' :5,’THE RESULT WAS TOO LARGE’,big:5,’ TIMES, AND’);
  writeln(' :10,’TOO SMALL’,small:5,’ TIMES’);
  writeln;
  writeln;
  writeln;
  writeln(' :5,’THE MAXIMUM RELATIVE ERROR’,big:5,’ TIMES’);
  writeln;
  writeln;
  writeln;
  writeln;
  writeln;
  writeln;
  writeln(‘GENERIC:’,lb,‘:’,ub,’:mean:15,’mean:15,’);
  writeln(' :10,’(lb,ub):15,’(lb,ub):15,’);
  writeln;
  writeln;
  writeln('
');
end;

function sign(a1, a2 : real) : real;
begin
  if (a2 < 0) then
    sign := -abs(a1)
  else
    sign := a1
end;
sign := abs(a);  
end;

begin  
  iy := 100001L;  
  m := Machar ( i beta , it , irnd , negexp , mexp ,  
                eps , epsneg , xmin , xmax );  
  beta := ( i beta );  
  albeta := ln ( beta );  
  j := it div 3;  
  a := 1.0;  

  for i := 1 to j do begin  
    a := a1beta;  
  end;

  n := 2000;  
  xn := ( n );  
  b := 1.0 + a;  
  a := 1.0 - a;  
  d := 1.0 + sqrt ( eps );  
  half := 0.5;  
  eight := 8.0;  
  tenth := 0.1;  
  il := 0;  

  for i := 1 to n do begin  
    xl := a / beta;  
    d := del * random * d + xl;  
    case j of  
      1: begin  
        write("TEST OF LN(X) VS TAYLOR SERIES EXPANSION");  
        writeln( 'OF LN(1+Y)' );  
        end;
      2: begin  
        writeln( 'TEST OF LN(X) VS LN(17X/16)-LN(17/16)' );  
        writeln( 'OF LN(X)^2' );  
        a := sqrt ( half );  
        b := 15.0 / 16.0;  
        end;
      3: begin  
        writeln( 'TEST OF LN(X) VS LN(11X/10)-LN(11/10)' );  
        writeln( 'OF LN(X)^3' );  
        a := sqrt ( tenth );  
        b := 0.9;  
        end;
      4: begin  
        writeln( 'TEST OF LN(X) VS 2 * LN(X)' );  
        writeln( 'OF LN(X)^4' );  
        a := 16.0;  
        b := 240.0;  
        end;
    end;

  end;  

  for i := 1 to n do begin  
    y := x - 0.5;  
    z := ln ( x );  
    zz := 1.0 / 3.0;  
    bb := 1.0 / 15.0;  
    w := ( z - half ) * y + y;  
    end;

  for i := 1 to n do begin  
    x := del * random * d + xl;  
    case j of  
      1: begin  
        begin  
          y := ( x - 0.5 ) - half;  
          z := ln ( x );  
          zz := 1.0 / 3.0;  
          bb := 1.0 / 15.0;  
          w := ( z - half ) * y + y;  
        end;
      2: begin  
        x := ( x + 0.8 ) - eight;  
        y := x + x / 16.0;  
        z := ln ( x );  
        zz := ln ( y ) - 7.77468164384258e-5;  
        end;
      3: begin  
        x := ( x + 0.8 ) - eight;  
        y := x + x / 16.0;  
        z := ln ( x );  
        zz := ln ( y ) - 7.77468164384258e-5;  
        end;
      4: begin  
        begin  
          y := x + x * x;  
          z := ln ( x );  
          zz := ln ( x ) + zz;  
        end;
      end;

  end;

  w := ( z - zz ) / z;  
  if ( w < 0.0 ) then k := k + 1;  
  if ( z < 0.0 ) then k1 := k1 + 1;  
  r5 := r5 + w;  
  w := abs ( w );  
  if ( w > r4 ) then begin  
    r5 := w;  
    xl := x;  
  end;

  r7 := r4 + w;  
  x1 := x + del;  

end;  

r5 := r5 / xn;  
r7 := sqrt ( r4 / xn );  
printtestrun(na,b,k,k1,r5,r7,x1,r7);  
end;  

[ special tests ]
writeln(' THE IDENTITY LN(X) = -LN(1/X) WILL BE TESTED');
writeln;
writeln(' X F(X) + F(1/X)');
writeln;
for i := 1 to 5 do begin
  x := random;
  x := x + x + 15.0;
  y := 1.0 / x;
  z := ln(x) + ln(y);
  writeln(x:15, z:15);
end;
writeln(' LN (XMAX) = LN(', x:15, ') ');
writeln;
writeln;
writeln(' LN (XMIN) = LN(' , x:15, ') ');
writeln;
writeln;
Y :=
writeln;
for i := 1 to 5 do begin
  x := random;
  x := x + x + 15.0;
  y := 1.0 / x;
  z := ln(x) + ln(y);
  writeln(x:15, z:15);
end;
writeln;
determine negep, epsneg
negep := it + 1;
a := one;
for i := 1 to negep do begin
a := a * beta;
end;
while ((one + a) - one = zero) do begin
a := a * beta;
negep := negep - 1;
end;
negep := -negep;
epsneg := a;
[ determine machep, eps
machep := negep;
while ((one + a) - one = zero) do begin
a := a * beta;
machep := machep + 1;
end;
eps := a;
[ determine ngrd
ngrd := 0;
if ((irnd = 0) and (((one + eps) * one - one) <> zero)) then
ngrd := 1;
[ determine iexp, minexp, xmax
loop to determine largest i such that
(1/beta)**i does not underflow
exit from loop is signal by an underflow
i := 0;
betain := one / beta;
z := betain;
underflo := false;
repeat begin
y := z;
z := y * y;
end;
check for underflow
if ((z * one = zero) or (abs(z) > y)) then begin
underflo := true;
end else begin
i := i + 1;
end;
end until underflo;
k := 1;
[ determine k such that (1/beta)**k does not underflow
first not k * 2**

negeps is the exponent on the smallest positive fl. pt. no. such that 1.0-negeps <> 1.0, except that negeps is 1.0 when no representations are possible.

iexp is the number of bits (decimal places if ibeta = 10) reserved for the representation of the exponent of a floating-point number.

minexp is the exponent of the smallest positive fl. pt. no.

maxexp is the exponent of the largest finite floating-point number. xmin is the smallest positive floating-point number. in particular, xmin = beta ** minexp

maxexp is the exponent of the largest finite floating-point number. in particular, xmax = (1.0-maxexp) * ibeta ** maxexp

eps is the smallest positive floating-point number such that 1.0-eps <> 1.0, except that the exponent neges is bounded below by it-3. In particular, eps = ibeta ** maxexp

epsneg is the smallest positive floating-point number such that 1.0-eps <> 1.0, except that the exponent negeps is bounded below by it-3. In particular, epsneg = ibeta ** negeps

xmin is the smallest positive floating-point number. in particular, xmin = ibeta ** minexp

xmax is the largest finite floating-point number. in particular, xmax = (1.0-epsneg) * ibeta ** maxexp

i, iz, j, k, mx : integer;

a, b, beta, betain, betaml, one, y, z, zero : real;

underflo : boolean;

begin

irnd := 1;
one := (irnd);
a := one + one;
b := a;
zero := 0.0;

[ determine ibeta, beta ala Malcolm

while ((a + one) - a = one) do begin
a := a + a;
end;
while (a + b) = a do begin
b := b + b;
end;
betain := trunc((a + b) - a);
beta := (ibeta);
betaml := beta - one;
[ determine irnd, ngrd, it

if ((a + betaml) - a = zero) then irnd := 0;
it := 0;
a := one;
repeat begin
it := it + 1;
a := a * beta;
end until ((a + one) - a = one <> zero);

{ determine negep, epsneg
negep := it + 1;
a := one;
for i := 1 to negep do begin
a := a * beta;
end;
while ((one + a) - one = zero) do begin
a := a * beta;
negep := negep - 1;
end;
negep := -negep;
epsneg := a;
[ determine machep, eps
machep := negep;
while ((one + a) - one = zero) do begin
a := a * beta;
machep := machep + 1;
end;
s := a;
[ determine ngrd
ngrd := 0;
if ((irnd = 0) and (((one + eps) * one - one) <> zero)) then
ngrd := 1;
[ determine iexp, minexp, xmax
loop to determine largest i such that
(1/beta)**i does not underflow
exit from loop is signal by an underflow
i := 0;
betain := one / beta;
z := betain;
underflo := false;
repeat begin
y := z;
z := y * y;
end;
check for underflow
if ((z * one = zero) or (abs(z) > y)) then begin
underflo := true;
end else begin
i := i + 1;
end;
end until underflo;
k := 1;
[ determine k such that (1/beta)**k does not underflow
first not k * 2**
for j := 1 to i do begin
  k := k + k;
end;

iexp := i + 1;
mx := k + k;
if (ibeta = 10) then begin
  for decimal machines only
  iexp := 2;
  iz := ibeta;
  while (k >= iz) do begin
    iz := iz * ibeta;
    iexp := iexp + 1;
  end;
  mx := iz + iz - 1;
  end;
underflo := false;
repeat begin
  loop to construct xmin
  exit from loop is signalled by an underflow
  xmin := y;
  y := y * betain;
  if (((y * one) = zero) or (abs(y) > xmin))
    then begin
      underflo := true;
      end else begin
    xmin := y;
    end;
  end until underflo;
  minexp := -k;
  [determine maxexp, xmax]
  if ((mx <= k + k - 3) and (ibeta <> 10)) then begin
    maxexp := maxexp + 1;
    mx := mx + mx;
    end;
  maxexp := mx + minexp;
  [/adjust for machines with implicit leading]
  bit in binary significand and machines with
  radix point at extreme right of significand]
  i := maxexp + minexp;
  if (ibeta = 2) and (i = 0) then maxexp := maxexp - 1;
  if (i > 20) then maxexp := maxexp - 3;
  xmax := one - epsneg;
  if (xmax * one <> xmax) then xmax := one - beta * epsneg;
  xmax := (xmax * betain * betain * betain) / xmin;
  i := maxexp + minexp + 3;
  if (i > 0) then begin
    for j := 1 to i do begin
      xmax := xmax * beta;
      end;
    end;
end;
This program checks the implementation of the transfer functions trunc and round. The compiler fails if the program does not compile and run.

```pascal
program t5p5p6p3d1(output);
var
  i, truncstatus, roundstatus : integer;
  j : real;
begin
  truncstatus:=O;
  roundstatus:=O;
  if (trunc(3.7)=3) and (trunc(-3.7)=-3) then
    truncstatus:=truncstatus+l
  else
    writeln(' FAIL ••• 6.6.6.3-1: !RUNC');
  if (round(3.7)=4) and (round(-3.7)=-4) then
    roundstatus:=roundstatus+l
  else
    writeln(' FAIL ••• 6.6.6.3-1: !RUNC ROUND');
  j:=0;
  for i=-333 to 333 do
    begin
      j:=j+i div 100;
      if j<0 then
        if (trunc(j-0.5)=round(j)) then
          begin
            truncstatus:=truncstatus+l;
            roundstatus:=roundstatus+l
          end
        else
          writeln(' FAIL ••• 6.6.6.3-1: TRUNC ROUND');
      else
        if (trunc(j+0.5)=round(j)) then
          begin
            truncstatus:=truncstatus+l;
            roundstatus:=roundstatus+l
          end
        else
          writeln(' FAIL ••• 6.6.6.3-1: TRUNC ROUND');
    end;
  if (truncstatus=668) and (roundstatus=668) then
    writeln(' PASS ••• 6.6.6.3-1');
end.
```

This program causes an error to occur as the result returned by the trunc function is not a value of the type integer. The error should be detected at run-time.

```pascal
program t6p6p6p3d2(output);
var
  reel real;
  integer;
  boolean; ok
begin
  reel:=1111.1111;
  ok:=true;
  while ok do
    begin
      i :=trunc (reel);
      if (i<0) then
        ok:=false
      else
        reel:=reel*2
    end;
  writeln(' ERROR NOT DETECTED ••• 6.6.6.3-2');
end.
```

This program causes an error to occur as the result returned by the round function is not a value of the type integer. The error should be detected at run-time.

```pascal
program t6p6p6p3d3(output);
var
  reel : real;
  i : integer;
  ok : boolean;
begin
  reel:=1111.1111;
  ok:=true;
  while ok do
    begin
      i:=round(reel);
      if (i<0) then
        ok:=false
      else
        reel:=reel*2
    end;
  writeln(' ERROR NOT DETECTED ••• 6.6.6.3-3');
end.
```
This test checks that neither trunc nor round are permitted to have integer parameters. The Standard requires these to be real. The compiler deviates if the program compiles and prints DEVIATES.

program t5p6p6p3d4(output);
var
  i: integer;
  x: real;
begin
  i := 1979;
  x := trunc(i) + round(i + 1);
write(' DEVIATES ... 6.6.6.3-4, TRUNC/ROUND')
end.

This program checks that the implementation of the ord function is as described by the Standard. The compiler fails if the program does not compile and run.

program t5p6p6p4dl(output);
type
colourtype = (red, orange, yellow, green, blue);
var
colour: colourtype;
some: integer;
ok: boolean;
begin
  counter := 0;
  if (ord(false) = 0) and (ord(true) = 1) then
    counter := counter + 1
  else
    writeln(' FAIL ... 6.6.6.4-1: FALSE/TRUE');
  if (ord(red) = 0) and (ord(orange) = 1) and
    (ord(yellow) = 2) and (ord(green) = 3) and
    (ord(blue) = 4) then
    counter := counter + 1
  else
    writeln(' FAIL ... 6.6.6.4-1: COLOURTYPE');
  i := 1;
  ok := true;
  while ok do
    begin
      i := i + 1;
      if i > 10 then
        ok := false
      else
        if ord(i) = i then
          counter := counter + 1
        else
          begin
            ok := false;
            writeln(' FAIL ... 6.6.6.4-1: I')
          end;
    end;
  colour := blue;
  some := orange;
  if ord(colour) = 4 then
    counter := counter + 1
  else
    writeln(' FAIL ... 6.6.6.4-1: COLOUR');
  if ord(some) = 1 then
    counter := counter + 1
  else
    writeln(' FAIL ... 6.6.6.4-1: SOME');
  if counter = 25 then
    writeln(' PASS ... 6.6.6.4-1')
end.
This program checks the implementation of chr. The compiler fails if the program does not compile and run.

program t6p6p6p4d2(output);
var
    letter : char;
    counter : integer;
begin
    counter:=0;
    for letter:='O' to '9'
    do
        if chr(ord(letter)\=letter then
            counter :=counter+1;
    if counter=10 then
        writeln('PASS...6.6.6.4-2')
    else
        writeln('FAIL...6.6.6.4-2')
end.

This program tests the function pred only. The user is referred to tests 6.4.2.2-4 and 6.4.2.3-2 for tests of succ. The compiler fails if the program does not compile and run.

program t6p6p6p4d3(output);
type
    colourtype = (red,orange,yellow,green,blue);
var
    colour : colourtype;
    counter: integer;
begin
    counter:=0;
    colour:=blue;
    colour:=pred(colour);
    colour:=pred(colour);
    colour:=pred(succ(colour));
    if colour=yellow then
        counter:=1
    else
        writeln('FAIL...6.6.6.4-2: COLOUR');
    if pred(-10)=-11 then
        counter:=counter+1
    else
        writeln('FAIL...6.6.6.4-2: -VE NUMBERS');
    if counter=2 then
        writeln('PASS...6.6.6.4-3')
end.

This program causes an error to occur as the function succ is applied to the last value of an ordinal type. The error should be detected by the compiler or at run-time.

program t6p6p6p4d4(output);
type
    enumerated = (first,second,third,last);
var
    ordinal : enumerated;
begin
    ordinal:=succ(last);
    writeln('ERROR NOT DETECTED...6.6.6.4-4')
end.

This program causes an error to occur as the function pred is applied to the first value of an ordinal type. The error should be detected by the compiler or at run-time.

program t6p6p6p4d5(output);
type
    enumerated = (first,second,third,fourth,last);
var
    ordinal : enumerated;
begin
    ordinal :=first;
    ordinal:=pred(ordinal);
    writeln('ERROR NOT DETECTED...6.6.6.4-5, PRED')
end.

This test checks that succ and pred cannot be applied to real values. The compiler deviates if the program compiler and prints DEVIATES.

program t6p6p6p4d6(output);
var
    x:real;
begin
    x:=0.3;
    if (succ(x)>x) and (pred(x)<x) then
        writeln('DEVIATES...6.6.6.4-6, REAL SUCC/PRED')
    else
        writeln('DEVIATES...6.6.6.4-6, MESS')
end.
This test evokes an error by pushing chr past the limits of the char type. It assumes that no char type has more than 10000+ord('0') values.

```pascal
program test6p6p5d7(output);
var
  i: 0..10000;
  c: char;
begin
  for i := 0 to 10000 do
    c := chr(i+ord('0'));
  writeln('ERROR NOT DETECTED...6.6.6.4-7, CHR');
end.
```

This program would have tested the function of the eof and eoln predicates. However, the test is carried out elsewhere, and the user is referred to tests 6.4.3.5-2 and 6.4.3.5-3.

```pascal
program test6p6p5d1;
begin
end.
```

This program tests the predicate odd. The compiler fails if the program does not compile or the program states that this is so.

```pascal
program test6p6p5d2(output);
var
  i, counter : integer;
function myodd(i: integer): boolean;
begin
  myodd := (abs(i mod 2) = 1);
end;
begin
  counter := 0;
  for i := -10 to 10 do
    if odd(i) then
      begin
        if myodd(i) then counter := counter+1
        end else
          begin
            if not myodd(i) then counter := counter+1
          end;
    if counter=21 then
      writeln('PASS...6.6.6.5-2')
    else
      writeln('FAIL...6.6.6.5-2')
  end.
```

This test checks that the function odd is restricted to integer parameters. The compiler deviates if the program compiles and prints Deviates.

```pascal
program test6p6p5d3(output);
var
  x: real;
begin
  x := 1.0;
  if odd(x) then
    writeln('DEVIATES...6.6.6.5-3, REAL')
  else
    writeln('DEVIATES...6.6.6.5-3, MESS')
end.
```

This program tests that the precedence of the boolean operators is as described in the Pascal Standard. The compiler fails if the program does not compile, or the program states that this is so.

```pascal
program test6p7pldl(output);
var
  a, b, c, w, x : boolean;
  counter: integer;
begin
  counter := 0;
  for a := false to true do
    for b := false to true do
      for c := false to true do
        begin
          w := (a and b) < c;
          x := c > b and a;
          if (w=x) then counter := counter+1;
        end;
  if (counter=8) then
    writeln('PASS...6.7.1-1')
  else
    writeln('FAIL...6.7.1-1')
end.
```
This program tests that the precedence of the arithmetic operators is as described by the Pascal Standard.

The compiler fails if the program does not compile, or the program states that this is so.

```pascal
program t6p7p1d2(output);
var
  a,b,c,d,e,f,g : integer;
  h,i,j,k,l,m,n : real;
begin
  a:=1;
  b:=2;
  c:=3;
  d:=4;
  e:=5;
  f :=a-b+c-d;
  g:=e-d div b*c;
  h:=1;
  i :=2;
  j :=3;
  k :=4;
  l :=5;
  m:=h/i*j/k;
  n:=i+k/i-3;j;
  if (f=-2) and (g=-1) and (l=-2) and (m<0.38) and (m>0.37) then
    writeln(' PASS...6.7.1-2')
  else
    writeln(' FAIL...6.7.1-2')
end.
```

This program checks the operation of the operators + - and *.

The compiler fails if the program does not compile, or the program states that this is so.

```pascal
program t6p7p2p2d1(output);
var
  i, j, counter : integer;
begin
  counter :=0;
  for i:=0 to 1 do
    for j:=1 to 4 do
      if (i-j) = (i div j)*j then
        counter:=counter+1;
    if counter=56 then
      writeln(' PASS...6.7.2.2-2')
    else
      writeln(' FAIL...6.7.2.2-2: DIV MOD')
end.
```

This program checks that DIV and MOD are implemented by the rule specified by the Pascal Standard.

The compiler fails if the program does not compile, or the program states that this is so.

```pascal
program t6p7p2p2d2(output);
var
  i, j, counter : integer;
begin
  counter :=0;
  for i:=0 to 1 do
    for j:=1 to 4 do
      if (i mod j) = (i-(i div j)*j) then
        counter:=counter+1;
  if counter=56 then
    writeln(' PASS...6.7.2.2-2')
  else
    writeln(' FAIL...6.7.2.2-2: DIV MOD')
end.
```

This program causes an error to occur as the second operand of the DIV operator is 0.

The error should be detected at run-time.

```pascal
program t6p7p2p2d3(output);
var
  i, j, k : integer;
begin
  i:=6;
  j:=0;
  k:=i div j; { an error as j=0 }
  writeln(' ERROR NOT DETECTED...6.7.2.2-3: ZERO DIVIDE (DIV)')
end.
```
{TEST 6.7.2.2-4, CLASS=QUALITY}

This program checks that constant and variable operands for DIV produce the same result, and if negative operands are permitted.

program t6p7p2p2d4(output);
var i, j, k, l, m, counter : integer;
begin
  { The next few statements may cause a run-time error. }
  writeln(' THIS PROGRAM ATTEMPTS DIVISION WITH NEGATIVE OPERANDS');
  counter := 0;
  j:=2;
  for i:= -10 to 10 do
    begin
      l:=i div j;
      m:= l div 2;
      if (l=m) then counter := counter+1;
      l:=i mod j;
      m:= l mod 2;
      if (l=m) then counter := counter+1;
      writeln(' ERROR NOT DETECTED...6.7.2.2-5')
    end;
  if counter = 63 then
    begin
      writeln(' DIVISION INTO NEGATIVE OPERANDS IMPLEMENTED AND '');
      writeln('consistent');
    end else
    begin
      writeln(' inconsistent division into negative operands');
      counter := 0;
      j:=2;
      for i:= -10 to 10 do
        begin
          l:=i div j;
          m:= l div 2;
          if (l=m) then counter := counter+1;
          l:=i mod j;
          m:= l mod 2;
          if (l=m) then counter := counter+1;
          writeln(' error not detected...6.7.2.2-6');
        end;
      if counter = 42 then
        begin
          writeln(' division by negative operands implemented and '');
          writeln('consistent');
        end else
        begin
          writeln(' inconsistent division by negative operands');
          i:=3;
          if (i div 2 = -1) then
            begin
              writeln(' quotient = trunc(a/b) for negative operands');
              writeln(' modulus returns remainder of div');
            end;
          if (l mod 2 = 1) then
            begin
              writeln(' modulus returns remainder of div');
            end;
          writeln(' error not detected...6.7.2.2-7');
        end.
end.

{TEST 6.7.2.2-5, CLASS=CONFORMANCE}

This program checks that maxint satisfies the conditions laid down in the Pascal Standard. The compiler fails if the program does not compile, or does not print pass.

program t6p7p2p2d5(output);
var i : integer;
begin
  i:=-maxint;
  if (odd(maxint)) then
    begin
      i:=(maxint-(maxint div 2)+1)*2;
      writeln(' error not detected...6.7.2.2-5: maxint');
    end;
  i:=(-maxint+(maxint div 2))*2-2;
  if (i<=maxint) then
    begin
      writeln(' error not detected...6.7.2.2-6: maxint');
      writeln(' pass...6.7.2.2-5: maxint');
    end;
  i:=(maxint-(maxint div 2)+1)*2;
  writeln(' pass...6.7.2.2-5: maxint');
end.

{TEST 6.7.2.2-6, CLASS=ERRORHANDLING}

This program causes an error to occur as the result of a binary integer operation is not in the interval 0 -> maxint.

program t6p7p2p2d6(output);
var i : integer;
begin
  i:=(maxint-(maxint div 2)+1)*2;
  writeln(' error not detected...6.7.2.2-5: maxint');
end.

{TEST 6.7.2.2-7, CLASS=ERRORHANDLING}

This program causes an error to occur as the result of a binary integer operation is not in the interval 0 -> maxint.

program t6p7p2p2d7(output);
var i : integer;
begin
  i:=(maxint-(maxint div 2)+1)*2;
  writeln(' error not detected...6.7.2.2-5: maxint');
end.
This program causes an error to occur as the second operand of the MOD operator is 0. The error should be detected at run-time.

```
program t6p7p2p2d8(output);
var
  i, j, k : integer;
begin
  i:=6;
  j:=0;
  k:=i mod j; { an error as j=0 }
  writeln(' ERROR NOT DETECTED...6.7.2.2-8: MOD ZERO')
end.
```

The unary operator plus can clearly only be applied to numeric operands. Hence this program should fail to compile. The compiler deviates if the program compiles and prints DEVIATES.

```
program t6p7p2p2d9(output);
const
capa = 'A';
begin
  writeln(capa);
  writeln(' DEVIATES...6.7.2.2-9, UNARY OPERATOR')
end.
```

This test checks the operation of the boolean operators. The compiler fails if the program does not compile, or the program states that this is so.

```
program t6p7p2p3dl(output);
var
  a,b,c : boolean;
  counter : integer;
begin
  counter:=0;
  a:=false;
  b:=false;
  { OR truth table }
  if a or b then writeln(' FAIL...6.7.2.3-1: OR')
  else begin
    b:=true;
    if a or b then begin
      counter:=counter+1
    end
    writeln(' FAIL...6.7.2.3-1: OR')
  end;
  { AND truth table }
  a:=false;
  b:=false;
  if a and b then writeln(' FAIL...6.7.2.3-1: AND')
  else begin
    b:=true;
    if a and b then
      writeln(' FAIL...6.7.2.3-1: AND')
    else begin
      a:=true;
      b:=false;
      if a and b then
        writeln(' FAIL...6.7.2.3-1: AND')
      else begin
        b:=true;
        if a and b then
          counter:=counter+1
      end;
end;
```

{ TEST 6.7.2.2-9, CLASS=DEVIANCE }
{ This test checks the operation of the boolean operators. The compiler deviates if the program does not compile, or the program states that this is so. }

```
program t6p7p2p3dl(output);
var
  a,b,c : boolean;
  counter : integer;
begin
  counter:=0;
  a:=false;
  b:=false;
  { OR truth table }
  if a or b then writeln(' FAIL...6.7.2.3-1: OR')
  else begin
    b:=true;
    if a or b then begin
      counter:=counter+1
    end
    writeln(' FAIL...6.7.2.3-1: OR')
  end;
  { AND truth table }
  a:=false;
  b:=false;
  if a and b thenwriteln(' FAIL...6.7.2.3-1: AND')
  else begin
    b:=true;
    if a and b then
      writeln(' FAIL...6.7.2.3-1: AND')
    else begin
      a:=true;
      b:=false;
      if a and b then
        writeln(' FAIL...6.7.2.3-1: AND')
      else begin
        b:=true;
        if a and b then
          counter:=counter+1
      end;
```

{ TEST 6.7.2.2-8, CLASS=ERRORHANDLING }
{ This program causes an error to occur as the second operand of the MOD operator is 0. The error should be detected at run-time. }

```
program t6p7p2p2d8(output);
var
  i, j, k : integer;
begin
  i:=6;
  j:=0;
  k:=i mod j; { an error as j=0 }
  writeln(' ERROR NOT DETECTED...6.7.2.2-8: MOD ZERO')
end.
```

{ TEST 6.7.2.2-9, CLASS=DEVIANCE }
{ The unary operator plus can clearly only be applied to numeric operands. Hence this program should fail to compile. The compiler deviates if the program compiles and prints DEVIATES. }

```
program t6p7p2p2d9(output);
const
capa = 'A';
begin
  writeln(capa);
  writeln(' DEVIATES...6.7.2.2-9, UNARY OPERATOR')
end.
```
{ TEST 6.7.2.3-2, CLASS=IMPLEMENTATION DEFINED }

{ This program determines if a boolean expression is partially evaluated if the value of the expression is determined before the expression is fully evaluated }

program t6p7p2p3d2(output);
var
a:boolean;
k,l:integer;
function sideeffect(var i:integer; b:boolean):boolean;
begin
i :=i+1;
sideeffect :=b;
end;
{NOTE: NOT is sometimes badly implemented by wordwise complementation, and for this reason the following two tests may fail.}
begin
if (not not false)=true then
counter:=counter+1
else
writeln(' FAIL ••• 6.7.2.3-1: NOT FALSE');
if (not not true)=false then
counter:=counter+1
else
writeln(' FAIL ••• 6.7.2.3-1: NOT TRUE');
e:=false;
a:=true;
b:=false;
if (a or b)=(b or a) then
counter:=counter+1
else
writeln(' FAIL ••• 6.7.2.3-1: COMMUTATION');
if (a or b)=((a or b or c) then
if (a and b)=((a and b or c) then
if not(a or b) and(not a) and(not b) then
if not (not a)= a then
else
writeln(' FAIL...6.7.2.3-1: INVERSION');
if counter=10 then
writeln(' PASS...6.7.2.3-1');
end.
end.
end.

{ TEST 6.7.2.3-2, CLASS=IMPLEMENTATION DEFINED }

{ This program determines if a boolean expression is partially evaluated if the value of the expression is determined before the expression is fully evaluated }

program t6p7p2p3d2(output);
var
a:boolean;
k,l:integer;
function sideeffect(var i:integer; b:boolean):boolean;
begin
i :=i+1;
sideeffect :=b;
end;
{NOTE: NOT is sometimes badly implemented by wordwise complementation, and for this reason the following two tests may fail.}
begin
if (not not false)=true then
counter:=counter+1
else
writeln(' FAIL ••• 6.7.2.3-1: NOT FALSE');
if (not not true)=false then
counter:=counter+1
else
writeln(' FAIL ••• 6.7.2.3-1: NOT TRUE');
e:=false;
a:=true;
b:=false;
if (a or b)=(b or a) then
counter:=counter+1
else
writeln(' FAIL ••• 6.7.2.3-1: COMMUTATION');
if (a or b)=((a or b or c) then
if (a and b)=((a and b or c) then
if not(a or b) and(not a) and(not b) then
if not (not a)= a then
else
writeln(' FAIL...6.7.2.3-1: INVERSION');
if counter=10 then
writeln(' PASS...6.7.2.3-1');
end.
This program determines if a boolean expression is partially evaluated if the value of the expression is determined before the expression is fully evaluated.

```pascal
program t6p7p2p3d3(output);
var
  a:boolean;
  k,l:integer;
function sideeffect(var i:integer; b:boolean):boolean;
begin
  i:=i+l;
  sideeffect:=b;
end;
begin
  writeln('TEST OF SHORT CIRCUIT EVALUATION OF (A OR B)');
  k:=0;
  l:=0;
  a:=sideeffect(k,true) or sideeffect(l,true);
  if (k=O) and (l=1) then
    writeln('SECOND EXPRESSION EVALUATED ••• 6.7.2.3-3');
  else
    if (k=1) and (l=0) then
      writeln('FIRST EXPRESSION EVALUATED ••• 6.7.2.3-3');
    else
      writeln('BOTH EXPRESSIONS EVALUATED ••• 6.7.2.3-3');
  writeln(' FAIL ••• 6.7.2.3-3');
end.
```

Logical operators are allowed to be performed on integers. The compiler deviates if the program compiles and prints `DEVIATES.`

```pascal
program t6p7p2p3d4(output);
var
  i,j:integer;
begin
  i:=1; j:=2;
  if i=1 and j then
    writeln('DEVIATES ••• 6.7.2.3-4, LOGICAL OPS.');
end.
```

This test checks that operations on overlapping sets are detected. An error should be detected by the compiler or produced at run time.

```pascal
program t6p7p2p4d1(output);
var
  a,d : set of 0..10;
  b,c : set of 5..15;
begin
  b:=[5,10];
  a:=[0,5,10];
  d:=a+b;
  writeln('ERROR NOT DETECTED ••• 6.7.2.4-1: OVERLAPPING SETS');
end.
```

This test checks the operation of set operators. The compiler fails if the program does not compile, or the program states that this is so.

```pascal
program t6p7p2p4d2(output);
var
  a,b,c,d:set of 0..10;
  counter: integer;
begin
  counter:=0;
  a:=[0,2,4,6,8,10];
  b:=[1,3,5,7,9];
  c:=[0,1,2,3,4,5,6,7,8,9,10];
  d:=[a,b];  { ok }
  if (d=b) then
    counter:=counter+1;
  if (d=a) then
    counter:=counter+1;
  if (d=c) then
    counter:=counter+1;
  if (d=b) then
    counter:=counter+1;
  if (d=a) then
    counter:=counter+1;
  if (d=c) then
    counter:=counter+1;
  if (counter=5) then
    writeln('PASS ••• 6.7.2.4-2, SET OPERATORS');
  else
    writeln('FAIL ••• 6.7.2.4-2, SET OPERATORS');
end.
```
This program checks the operations of set operators on sets of constants and variables. The compiler fails if the program does not compile or the program states that this is so.

```
program t6p7p2p4d3(output);
var
  a,b,c: set of 0..10;
  counter: integer;
begin
  counter:=0;
  a:=[0,2,4,6,8,10];
  b:=[1,3,5,7,9];
  c:= [0,1,2,3,4,5,6,7,8,9,10];
  if (a+b=c) then
    counter:=counter+1;
  if (a-[] =a) then
    counter:=counter+1;
  if (c-a=b) then
    counter:=counter+1;
end.
```

This program tests the use of relational operators on strings. The operators denote lexicographic ordering according to the ordering of the character set. The compiler fails if the program does not compile, or the program states that this is so.

```
program t6p7p2p5d1(output);
type
  string=packed array[1..7] of char;
var
  string1, string2: string;
begin
  string1:='STRING1';
  string2:='STRING2';
  if (string1<>string2) then
    writeln('PASS ••• 6.7.2.5-1');
end.
```
This test checks the use of relational operators on sets. The compiler fails if the program does not compile, or the program states that this is so.

program t6p7p2p5d2(output);
var
a,b:set of 0..10;
c,counter:integer;
begin
  counter:=0;
  a:=[0,1,2,3,4,5];
  b:=[2,3,4];
  c:=3;
  if(a=[0,1,2,3,4,5]) then
    counter:=counter+1;
  if (a<>b) then
    counter :=counter+1;
  if(b<>[1,2,3,4,5]) then
    counter:=counter+l;
  if(b<=a) then
    counter:=counter+ 1;
  if(a>=b) then
    counter:=counter+l;
  if([0,1]<=a) then
    counter:=counter+l;
  if([1,2,3,4,5,6,10]>=b) then
    counter:=counter+l;
  if(1 in a) then
    counter:=counter+l;
  if(c in b) then
    counter:=counter+l;
  if(counter=9) then
    writeln(' PASS ••• 6.7.2.5-2, SET RELATIONAL OPERATORS');
  else
    writeln(' FAIL ••• 6.7.2.5-2, SET RELATIONAL OPERATORS');
end.

This test checks that file comparisons are not allowed. The semantics of this situation are particularly ill-defined, and not within standard Pascal. The compiler deviates if the program compiles and prints DEVIATES.

program t6p7p2p5d3(output);
var
  f:text;
begin
  rewrite(f);
  if f=output then
    writeln(' FAIL1...6.7.2.5-3, CONTENTS COMPARED');
  else
    writeln(' FAIL2...6.7.2.5-3, DESCRIPTORS COMPARED');
end.

Are relational operators permitted to concatenate? The compiler deviates if the program compiles and prints DEVIATES.

program t6p7p2p5d4(output);
var
  x,y,z:integer;
  b:boolean;
begin
  x:=1;
  y:=2;
  z:=3;
  b=(x<y<z);
  writeln(' DEVIATES...6.7.2.5-4, REL. OPS.');
end.
Does the compiler allow all the possible empty clauses?
The compiler fails if the program does not compile and print PASS.

program t5p8p2p2d1(output);
var
  b: boolean;
  r1: record
    x: real;
    a: integer;
  end;
  r2: record
    case b: boolean of
      true:
        c: real;
        d: char;
      false:
        end;
  end;
begin
  b := true;
  (e: integer);
  if b then
  if b then else
  repeat
    b := not b;
  until b;
  while b do
  begin
    b := not b;
  end;
  with r1 do
  r1.a := 1;
  case r1.a of
    0: b := false;
    1: ;
    2: b := true;
  end;
  writeln(' PASS...6.8.2.1-1, EMPTY STATEMENT');
  end.

This program determines whether selection of a variable involving
the indexing of an array occurs before or after the evaluation
of the expression in an assignment statement.

program t6p8p2p2d1(output);
var
  i: integer;
  a: array[1..3] of integer;
function sideeffect(var i: integer): integer;
begin
  sideeffect := i
end;
begin
  i := 1;
  a[1] := 0;
  a[2] := 0;
  a[1] := sideeffect(i);
  if a[1] = 2 then
  writeln(' SELECTION THEN EVALUATION...6.8.2.2-1')
  else
  if a[2] = 2 then
  writeln(' EVALUATION THEN SELECTION...6.8.2.2-1')
end.
This program is similar to 6.8.2.2-1, except that the selection of the variable involves the dereferencing of a pointer.

program t6p8p2p2d2(output);
type
  rekord = record
    a : integer;
    b : boolean;
    link : rekord
  end;
  poynter = rekord;
var
  temp, ptr : poynter;
function sideeffect(var p : poynter)
begin
  p := pT .link;
  sideeffect := 2;
end;
begin
  writeln(' TEST OF BINDING ORDER (pT := EXP)');
  new(ptr);
  ptrT .a := l;
  ptrT .b := true;
  new(temp);
  ptrT .link := temp;
  tempT .a := O;
  tempT .b := false;
  temp := ptr;
  ptrT .a := sideeffect (ptr);
  if tempT .a = 2 then
    writeln(' SELECTION THEN EVALUATION ... 6.8.2.2-2')
  else
    writeln(' EVALUATION THEN SELECTION ... 6.8.2.2-2')
end.

[TEST 6.8.2.4-2, CLASS=DEVIANCE]
[This test checks whether jumps between branches of an if statement are allowed. The compiler deviates if the program compiles and the program prints "DEVIATES." ]

program t6p8p2p4d2(output);
label 1, 2;
var
  i: integer;
begin
  i := 5;
  if (i < 10) then
    writeln(' DEVIATES ... 6.8.2.4-2');
  else
    2: writeln(' DEVIATES ... 6.8.2.4-2');
  end.
This test checks whether jumps between branches of a case statement are allowed. The compiler deviates if the program compiles and the program prints \textit{DEVIATES}.

```pascal
program t6p8p2p4d3(output);
label 4;
var
  i: 1..3;
begin
  for i:=1 to 2 do
    case i of
      1: ,
      2: goto 4;
      3: 4:
        writeln(' \textit{DEVIATES}...6.8.2.4-3');
    end;
end.
```

This test checks that a goto statement causes an error when the statement(S) to which control is transferred is not activated either by S or a statement in the statement sequence of which S is an immediate constituent. The compiler deviates if the compiler prints \textit{DEVIATES}.

```pascal
program t5p8p2p4d4(output);
var
  flag :boolean;
procedure a(i:integer;b:boolean);
label 99;
procedure r:
begin
  goto 99;
end;
begin
  case i of
    0:99:
      if (b) then
        writeln(' \textit{DEVIATES}...6.8.2.4-4')
      else
        if flag then
          writeln(' \textit{PASS}...6.8.2.4-4')
        else begin
          flag := true;
          a(1,false);
        end;
    1:
      a(2,true);
    2:
      r;
  end;
end;
begin
  flag := false;
  a(0,false);
end.
```

```
```
This test checks a nested if statement whose syntax is apparently ambiguous. The compiler fails if the program does not compile or the program states this by writing FAIL.

```
program t5p8p3p4d1(output);
const
  off=false;
var
  b:boolean,
begin
  for b:=false to true do
    begin
      if b then
        if off then
          writeln(' FAIL ••• 6.8.3.4-1')
        else
          end,
      begin
        if not b then
          writeln(' FAIL ••• 6.8.3.4-1')
        else
          writeln(' PASS ••• 6.8.3.4-1'),
        end;
    end.
end.
```

This test checks that a minimal case statement will compile. The compiler fails if the program does not compile.

```
program t5p8p3p5d1(output),
type
day=(mon,tue,wed),
var
  i :integer,
begin
  i:=1;
  case i of
    1:
    begin
      writeln(' PASS...6.8.3.5-1, CASE');
    end.
end.
```

This test checks that the case constants are of the same type as the case index. A compiler of good quality will detect that one path of the case statement cannot be taken and issue a warning message. The case-index in this test is a subrange and the case-constants are of the base type of the subrange.

```
program t5p8p3p5d2(output) ,
type
day=(mon,tue,wed),
var
  a: integer;
  d:mon •• tue,
begin
  for d:=mon to tue do
    case d of
      mon:
        a:=1,
      tue:
        a:=2,
      wed:
        a:=3,
        writeln(' QUALITY TEST - WARNINGS FOR IMPOSSIBLE CASES');
    end;
  writeln(' PASS ••• 6.8.3.5-2, CASE CONSTANTS'),
end.
```

This test checks that the constants of a case statement cannot be strings. The compiler deviates if the program compiles and prints DEVIATES.

```
program t5p8p3p5d3(output) ,
var
  a:char;
  i:integer;
begin
  for a='a' to 'd' do
    case a of
      'a': i:=1;
      'b': i:=i+1;
      'c': i:=i+1;
      'd': i:=i+1;
    end;
  writeln(' DEVIATES...6.8.3.5-3, CASE');
end.
```
This test checks that a compiler handles a sparse case adequately. Most compilers issue a jump table for a case, regardless of its structure. It is easy to optimise case statements to generate conditional statements if this is more compact. The compiler fails if the program does not compile or the program fails in execution.

```pascal
program t6p8p3p5d4(output);
var
  i, j: integer;
begin
  i := 1000;
  for j := 1 to 2 do
    case i of
      -1000: i := -i;
      1000: writeln(' PASS...6.8.3.5-4, SPARSE CASE');
    end;
  writeln(' ERROR NOT DETECTED...6.8.3.5-4, CASE');
end.
```

This test checks the type of error produced when the case statement does not contain a constant of the selected value. An execution error should be produced.

```pascal
program t6p8p3p5d5(output);
var
  i: integer;
begin
  i := 0;
  case i of
    -3, 3: writeln(' FAIL...6.8.3.5-5, CASE');
    1..2: writeln(' DEVIATES...6.8.3.5-5, CASE');
    end;
  writeln(' ERROR NOT DETECTED...6.8.3.5-5, CASE CONSTANT');
end.
```

This test is similar to the previous one - a case statement is given without a case-constant of the selected value. An error should be produced at execution time.

```pascal
program t6p8p3p5d6(output);
var
  i: integer;
begin
  i := 1000;
  case i of
    -3, 3: writeln(' FAIL...6.8.3.5-6, CASE');
    end;
  writeln(' ERROR NOT DETECTED...6.8.3.5-6, CASE CONSTANT');
end.
```
{TEST 6.8.1.5-8, CLASS=QUALITY}
{ This test checks a large populated case statement to check the
limit on the size of code is not a serious one.
The compiler has a small limit on the size of the case
statement if the program does not compile and print PASS. }

program tsp8p3p5d8(output);
var
  sum:integer;
i:0..255;
begin
  sum := 0;
  for i:=0 to 255 do case i of
  0 : sum := sum + 1;
  1 : sum := sum + 1;
  2 : sum := sum + 1;
  3 : sum := sum + 1;
  4 : sum := sum + 1;
  5 : sum := sum + 1;
  6 : sum := sum + 1;
  7 : sum := sum + 1;
  8 : sum := sum + 1;
  9 : sum := sum + 1;
  10 : sum := sum + 1;
  11 : sum := sum + 1;
  12 : sum := sum + 1;
  13 : sum := sum + 1;
  14 : sum := sum + 1;
  15 : sum := sum + 1;
  16 : sum := sum + 1;
  17 : sum := sum + 1;
  18 : sum := sum + 1;
  19 : sum := sum + 1;
  20 : sum := sum + 1;
  21 : sum := sum + 1;
  22 : sum := sum + 1;
  23 : sum := sum + 1;
  24 : sum := sum + 1;
  25 : sum := sum + 1;
  26 : sum := sum + 1;
  27 : sum := sum + 1;
  28 : sum := sum + 1;
  29 : sum := sum + 1;
  30 : sum := sum + 1;
  31 : sum := sum + 1;
  32 : sum := sum + 1;
  33 : sum := sum + 1;
  34 : sum := sum + 1;
  35 : sum := sum + 1;
  36 : sum := sum + 1;
  37 : sum := sum + 1;
  38 : sum := sum + 1;
  39 : sum := sum + 1;
  40 : sum := sum + 1;
  41 : sum := sum + 1;
  42 : sum := sum + 1;
  43 : sum := sum + 1;
  44 : sum := sum + 1;
  45 : sum := sum + 1;
end.
107: sum := sum + i;
108: sum := sum + i;
109: sum := sum + i;
110: sum := sum + i;
111: sum := sum + i;
112: sum := sum + i;
113: sum := sum + i;
114: sum := sum + i;
115: sum := sum + i;
116: sum := sum + i;
117: sum := sum + i;
118: sum := sum + i;
119: sum := sum + i;
120: sum := sum + i;
121: sum := sum + i;
122: sum := sum + i;
123: sum := sum + i;
124: sum := sum + i;
125: sum := sum + i;
126: sum := sum + i;
127: sum := sum + i;
128: sum := sum + i;
129: sum := sum + i;
130: sum := sum + i;
131: sum := sum + i;
132: sum := sum + i;
133: sum := sum + i;
134: sum := sum + i;
135: sum := sum + i;
136: sum := sum + i;
137: sum := sum + i;
138: sum := sum + i;
139: sum := sum + i;
140: sum := sum + i;
141: sum := sum + i;
142: sum := sum + i;
143: sum := sum + i;
144: sum := sum + i;
145: sum := sum + i;
146: sum := sum + i;
147: sum := sum + i;
148: sum := sum + i;
149: sum := sum + i;
150: sum := sum + i;
151: sum := sum + i;
152: sum := sum + i;
153: sum := sum + i;
154: sum := sum + i;
155: sum := sum + i;
156: sum := sum + i;
157: sum := sum + i;
158: sum := sum + i;
159: sum := sum + i;
160: sum := sum + i;
161: sum := sum + i;
162: sum := sum + i;
163: sum := sum + i;
164: sum := sum + i;
165: sum := sum + i;
166: sum := sum + i;
167: sum := sum + i;
168: sum := sum + i;
169: sum := sum + i;
170: sum := sum + i;
171: sum := sum + i;
172: sum := sum + i;
173: sum := sum + i;
174: sum := sum + i;
175: sum := sum + i;
176: sum := sum + i;
177: sum := sum + i;
178: sum := sum + i;
179: sum := sum + i;
180: sum := sum + i;
181: sum := sum + i;
182: sum := sum + i;
183: sum := sum + i;
184: sum := sum + i;
185: sum := sum + i;
186: sum := sum + i;
187: sum := sum + i;
188: sum := sum + i;
189: sum := sum + i;
190: sum := sum + i;
191: sum := sum + i;
192: sum := sum + i;
193: sum := sum + i;
194: sum := sum + i;
195: sum := sum + i;
196: sum := sum + i;
197: sum := sum + i;
198: sum := sum + i;
199: sum := sum + i;
200: sum := sum + i;
201: sum := sum + i;
202: sum := sum + i;
203: sum := sum + i;
204: sum := sum + i;
205: sum := sum + i;
206: sum := sum + i;
207: sum := sum + i;
208: sum := sum + i;
209: sum := sum + i;
210: sum := sum + i;
211: sum := sum + i;
212: sum := sum + i;
213: sum := sum + i;
214: sum := sum + i;
215: sum := sum + i;
216: sum := sum + i;
217: sum := sum + i;
218: sum := sum + i;
219: sum := sum + i;
220: sum := sum + i;
221: sum := sum + i;
222: sum := sum + i;
223: sum := sum + i;
224: sum := sum + i;
225: sum := sum + i;
226: sum := sum + i;
227: sum := sum + i;
228: sum := sum + i;
var

begin

end.

program t6p8p3p5d10(output);
var
  i, counter: integer;
  r: real;

begin
  counter := 0;
  for i := 1 to 4 do
    case r of
      1.0: counter := counter + 1;
      2.0: counter := counter + 1;
      3.0: counter := counter + 1;
      4.0: counter := counter + 1;
    end;
  end;

  if counter = 4 then
    writeln(' DEVIANCE...' , 6.8.3.5-11, CASE = DEVIANCE)
  else
    writeln(' FAILS...6.8.3.5-11, CASE CONSTANTS');

end.

{TEST 6.8.3.5-11, CLASS = DEVIANCE}

[ This test checks that the compiler detects a case index and the case constants are of different types. The compiler deviates if the program compiles and the program prints deviates. ]

program t6p8p3p5d01(output);
var
  i, counter: integer;
  r: real;

begin
  counter := 0;
  for i := 1 to 4 do
    case i of
      1: counter := counter + 1;
      2: counter := counter + 1;
      3: counter := counter + 1;
      4: counter := counter + 1;
    end;
  end;

  if counter = 4 then
    writeln(' DEVIATES...' , 6.8.3.5-9, CASE CONSTANTS')
  else
    writeln(' FAILS...6.8.3.5-9, CASE CONSTANTS');

end.

{TEST 6.8.3.5-9, CLASS = DEVIANCE}

[ This test checks that the compiler detects case-constants and the case-index are of different types. The compiler deviates if the program compiles and the program prints deviations. ]

program t6p8p3p5d9(output);
var
  i, counter: integer;

begin
  counter := 0;
  for i := 1 to 4 do
    case i of
      1: counter := counter + 1;
      2: counter := counter + 1;
      3: counter := counter + 1;
      4: counter := counter + 1;
    end;

  if counter = 4 then
    writeln(' DEVIATES...6.8.3.5-9, CASE CONSTANTS')
  else
    writeln(' FAILS...6.8.3.5-9, CASE CONSTANTS');

end.

{TEST 6.8.3.5-10, CLASS = DEVIANCE}

[ This test checks that the compiler detects real case constants and a real case index, even when the values are integers. The compiler fails if the program compiles and the program prints FAILS. ]

program t6p8p3p5d10(output);
var
  i, counter: integer;
  r: real;

begin
  counter := 0;
  for i := 1 to 4 do
    case r of
      1.0: counter := counter + 1;
      2.0: counter := counter + 1;
      3.0: counter := counter + 1;
      4.0: counter := counter + 1;
    end;

  if counter = 4 then
    writeln(' DEVIANCE...6.8.3.5-10, CASE CONSTANTS')
  else
    writeln(' FAILS...6.8.3.5-10, CASE CONSTANTS');
Some processors allow subrange-like lists to be used as case-constant elements. This test checks to see if this is allowed. It is not standard Pascal. The compiler deviates if the program compiles and prints DEVIATES.

program t5p8p3p5d12(output);
var
  thing:(a,b,c,d,e,f);
begin
  thing:=a;
  while thing<>f do begin
    case thing of
      a,d: thing:=succ(thing);
      e: thing:=f
    end;
  end;
  writeln(' DEVIATES...6.8.3.5-12, CASE CONSTANTS')
end.

Similar to test 6.8.3.5-12, this test checks the subrange case extension, which may not be safely implemented. The compiler deviates if the program compiles and prints DEVIATES.

program t5p8p3p5d13(output);
var
t,thing:(a,b,c,d,e,f,g,h);
begin
  for thing:=a to g do begin
    case thing of
      a..e: t:=thing;
      d..g: t:=succ(thing);
      b: t:=pred(thing)
    end;
  end;
  writeln(' DEVIATES...6.8.3.5-13, CASE CONSTANTS')
end.

This test checks whether an otherwise clause in a case statement is accepted. The convention is that adopted at the UCSD Pascal workshop in July 1978. The extension is accepted if the program compiles and prints EXTENSION - PASS.

program t5p8p3p5d14(output);
var
  i,j,k,counter:integer;
begin
  counter:=0;
  for i:=0 to 10 do
    case i of
      1,3,5,7,9:
        counter:=counter+1;
      otherwise
        j:=counter;
        k:=j;
    end;
  end;
  if (counter = 5) then
    writeln(' EXTENSION - PASS...6.8.3.5-14, OTHERWISE')
  else
    writeln(' EXTENSION - FAIL...6.8.3.5-14, OTHERWISE');
end.

This test checks that a repeat loop is executed at least once. The compiler fails if the program prints FAILS.

program t5p8p3p7d1(output);
var
counter: integer;
bool:boolean;
begin
  bool:=true;
  counter:=0;
  repeat
    counter:=counter+1
  until bool;
  if (counter=1) then
    writeln(' PASS...6.8.3.7-1, REPEAT')
  else
    writeln(' FAIL...6.8.3.7-1, REPEAT');
end.
The compiler fails if the program does not compile or the program prints FAIL.

program t5p8p3p7d2(output);
var
  a:integer;
function bool : boolean;
begin
  a:=a+1;
  bool := a>=5;
end;
begin
  a:=0;
  repeat
    until bool;
    if (a=5) then
      writeln(' PASS...6.8.3.7-2, EMPTY REPEAT')
    else
      writeln(' FAIL...6.8.3.7-2, EMPTY REPEAT');
  end.
end.

The compiler fails if the initial value of the boolean expression is false.

program t5p8p3p8d1(output);
var
  bool :boolean;
  counter: integer;
begin
  counter:=0;
  bool:=false;
  while bool do
  begin
    counter:=counter+1;
    bool:=false;
  end;
  if (counter=0) then
    writeln(' PASS...6.8.3.8-1, WHILE')
  else
    writeln(' FAIL...6.8.3.8-1, WHILE');
end.

The compiler fails if the program prints FAIL.

program t5p8p3p8d2(output);
var
  a:integer;
function bool:boolean;
begin
  a:=a+1;
  bool:=(a>=5);
end;
begin
  a:=0;
  while not bool do 
    if (a=5) then
      writeln(' PASS...6.8.3.8-2, EMPTY WHILE')
    else
      writeln(' FAIL...6.8.3.8-2, EMPTY WHILE');
  end.
end.
This program checks that assignment follows the evaluation of both expressions in a for statement. The compiler fails if the program prints FAIL.

Program t6p8p3p9d1(output);
var i,j:integer;
begn
i:=1;
j:=0;
for i:= (i+1) to (i+10) do
begin
j:=j+1;
writeln(i);
end;
if (j=10) then
writeln(' PASS...6.8.3.9-1, FOR')
else
writeln(' FAIL...6.8.3.9-1, FOR');
end.

This program checks that an assignment cannot be made to a for statement control variable. The compiler deviates if the program compiles and prints DEVIATES.

Program t6p8p3p9d2(output);
var i,j:integer;
begn
j:=0;
for i:=1 to 10 do
begin
j:=j+1;
i:=i+1;
writeln(j,i);
end;
writeln(' DEVIATES...6.8.3.9-2, FOR');
end.

This test checks that an error is produced when an assignment is made to a for statement control variable. The compiler deviates if the program compiles and prints DEVIATES.

Program t6p8p3p9d3(output);
var i,j:integer;
procedure nasty (var n:integer);
begin
n:=n+1;
end;
begn
j:=0;
for i:=1 to 10 do
begin
j:=j+1;
nasty(i);
end;
writeln(' DEVIATES...6.8.3.9-3, FOR');
end.

This program tests that an error is produced when an assignment is made to a for statement control variable. The compiler deviates if the program compiles and prints DEVIATES.

Program t6p8p3p9d4(output);
var i,j:integer;
procedure verynasty;
begin
i:=i+1;
end;
begn
j:=0;
for i:=1 to 10 do
begin
j:=j+1;
verynasty;
end;
writeln(' DEVIATES...6.8.3.9-4, FOR');
end.
This test checks that the use of a for statement control variable after the completion of the for statement, and without an intervening assignment is detected.

program t6p8p3p9d5(output);
var
  i,j,k,m:integer;
begin
  i:=100;
  j:=1;
  k:=10;
  m:=0;
  for i:=j to k do
    begin
      m:=m+1;
    end;
  writeln(' THE VALUE OF I =',i);
  writeln(' ERROR NOT DETECTED ••• 6.8.3.9-5, FOR');
end.

This program uses a for statement control variable after a for loop which is not entered. The control variable should be undefined after the for statement.

program t6p8p3p9d6(output);
var
  i,j,k,m:integer;
begin
  i:=100;
  k:=1;
  m:=0;
  j:=10;
  for i:=j to k do
    begin
      m:=m+1;
    end;
  writeln(' THE VALUE OF I =',i);
  writeln(' ERROR NOT DETECTED ••• 6.8.3.9-5, FOR');
end.

This test checks that extreme values may be used in a for loop. This will break a simply implemented for loop. In some compilers the succ test may fail at the last increment and cause wraparound(overflow) - leading to an infinite loop.

program t6p8p3p9d7(output);
var
  i,j:integer;
begin
  j:=0;
  for i:=(maxint-10) to maxint do
    j:=j+1;
  for i:=(-maxint+10) downto -maxint do
    j:=j+1;
  if j = 22 then
    writeln(' PASS ••• 6.8.3.9-7, FOR LOOP')
  else
    writeln(' FAIL ••• 6.8.3.9-7, FOR LOOP');
end.

This program checks that a control variable of a for statement is not undefined if the for statement is left via a goto statement. The compiler fails if the program does not compile or the program prints FAIL.

program t6p8p3p9d8(output);
label 100;
var
  i,j:integer;
begin
  j:=1;
  for i:=1 to 10 do
    begin
      if (j=5) then
        goto 100;
      j:=j+1;
    end;
  if i=j then
    writeln(' PASS ••• 6.8.3.9-8, FOR')
  else
    writeln(' FAIL ••• 6.8.3.9-8, FOR');
end.
program topBp3p9d9(output);
procedure p;
var
  i: integer;
procedure loop;
var
  j: integer;
begin
  j:=0;
  for i:=1 to 10 do
    j:=j+1;
end;
begin
  loop
end;
begin
  p;
  writeln(' DEVIATES...6.8.3.9-9, FOR');
end.

program topBp3p9d10(output);
var
  i: integer;
  counter: integer;
begin
  counter:=0;
  for i:=0.0 to 3.5 do
    counter:=counter+1;
if(counter=4) then
  writeln(' DEVIATES...6.8.3.9-10, FOR EXPRESSION ROUNDED')
else
  writeln(' DEVIATES...6.8.3.9-10, FOR EXPRESSION TRUNCATED');
end.

program topBp3p9d11(output);
var
  rec:record
    i, j: integer;
  end;
begin
  for rec.i:=0 to 10 do
    rec.j := rec.i;
  with rec do
    for i:=0 to 10 do
      j:=i;
  writeln(' DEVIATES...6.8.3.9-11, FOR');
end.

program topBp3p9d12(output);
type
  var
    Int = integer;
    ptr: int;
    j: integer;
begin
  j:=0;
  new(ptr);
  for ptrT := 0 to 10 do
    j:=j+1;
  writeln(' DEVIATES...6.8.3.9-12, FOR');
end.
This program tests whether a formal parameter can be used as a for statement control variable. The program deviates if the program compiles and prints DEVIATES.

```pascal
program t5p8p3p9d13(output);
  procedure p;
  var
    i: integer;
  procedure loop(var i:integer);
    var
      j: integer;
    begin
      j:=0;
      for i:=1 to 10 do
        j:=j+1;
    end;
    begin
      i:=10;
      loop(i)
    end;
  begin
    p;
    writeln('DEVIATES...6.8.3.9-13, FOR');
  end.
```

This program tests whether a global variable (at program level) can be used as a for statement control variable. The program deviates if the program compiles and prints DEVIATES.

```pascal
program t5p8p3p9d14(output);
  var
    i: integer;
  procedure p;
    procedure loop;
      var
        j: integer;
      begin
        j:=0;
        for i:=1 to 10 do
          j:=j+1;
      end;
      begin
        loop
      end;
    begin
      p;
      writeln('DEVIATES...6.8.3.9-14, FOR');
    end.
```

This program checks the order of evaluation of the limit expressions in a for statement. The compiler fails if the program prints FAIL.

```pascal
program t5p8p3p9d15(output);
  var
    i,j,k: integer;
  function f(var k:integer) : integer;
    begin
      k:=k+1;
      f:=f;
    end;
    begin
      k:=0;
      j:=0;
      for i:=f(k) to f(k)+10 do
        begin
          j:=j+1;
          writeln(i);
        end;
      if (j=12) then
        writeln('PASS...6.9.1.9-15, FOR')
      else
        writeln('FAIL...6.9.1.9-15, FOR');
    end.
```

[Test 6.8.3.9-13, CLASS=DEVIANCE]

[Test 6.8.3.9-14, CLASS=DEVIANCE]

[Test 6.8.3.9-15, CLASS=CONFORMANCE]
This test checks the type of error produced when a for statement control variable value is read during the execution of the for statement. The compiler deviates if the program compiles and prints DEVIATES.

program t6p8p3p9d16(output,f);
var
  f:text;
  i,j:integer;
begin
  j:=0;
  rewrite(f);
  writeln(f,5,5,5,5,5);
  reset (f);
  for i := 1 to 10 do
  begin
    if i=5 then
      read(f,i);
    j := j+1;
  end;
  writeln( ' DEVIATES ');
end.

This test checks the type of error produced when two nested for statements use the same control variable.

program t6p8p3p9d17(output);
var
  i,j,k:integer;
  m: integer;
begin
  j:=0;
  for i:=1 to 10 do
  begin
    m:=m+1;
  end;
  writeln(' THE UNDEFINED ORDINAL VALUE OF I IS ',ord(i));
  writeln(' ITS SYMBOLIC VALUE IS ');
end.

This test checks that the undefined state of a for-statement controlled variable when the loop is left has one or both of the following properties:
(a) Range checks are not omitted on these variables in the supposition that its value is permissible, or
(b) the value of the variable is in range of its type (in this specific implementation).
This test is not relevant if the use of the variable is prohibited.

program t6p8p3p9d18(output);
type
  t=(red,green,blue,pink);
var
  i,j,k:integer;
  m: integer;
begin
  i:=green;
  j:=red;
  k:=pink;
  m:=0;
  for i:=j to k do
  begin
    m:=m+1;
  end;
  writeln( ' JUST IN CASE THE RANGE ISNT CHECKED');
end.
This test checks that compilers that permit the deviation (extension?) of allowing non-local control variables do so responsibly and do not introduce new insecurities. This test checks that a nested for statement using the same control variable is detected. It is similar to test 6.8.3.9-14 but requires a degree of sophistication to detect this condition. The compiler deviates if the program prints DEVIATES. The program may loop endlessly under some compilers.

```
program t6p8p3p9d19(output);
var
  i : integer;
procedure p;
procedure q;
procedure r;
procedure s(var i:integer);
begin
  writeln(i);
end;
begin
  for i:= 5 downto 2 do 
    s(i);
end;
begin
  r
end;
begin
  for i:= 1 to 6 do 
    writeln(' DEVIATES...6.8.3.9-19, FOR')
  p;
  writeln(' DEVIATES...6.8.3.9-19, FOR')
end.
```

This test checks that for statements may be nested to 15 levels. The test may detect a small compiler limit, particularly those compilers that use a register for a control variable.

```
program t6p8p3p9d20(output);
var
  il,i2,i3,i4,i5,i6,i7,i8,i9,il0,il1,i12,i13,i14,i15:integer;
  j:integer;
begin
  for il:=1 to 2 do 
    for i2:=1 to 2 do 
      for i3:=1 to 2 do 
        for i4:=1 to 2 do 
          for i5:=1 to 2 do 
            for i6:=1 to 2 do 
              for i7:=1 to 2 do 
                for i8:=1 to 2 do 
                  for i9:=1 to 2 do 
                    for i10:=1 to 2 do 
                      for i11:=1 to 2 do 
                        for i12:=1 to 2 do 
                          for i13:=1 to 2 do 
                            for i14:=1 to 2 do 
                              for i15:=1 to 2 do 
                                j:=10;
  writeln(' FOR STATEMENT NESTED TO > 15 LEVELS...6.8.3.9-20')
end.
```
This program checks the implementation of the with statement. The compiler fails if the program does not compile or it compiles and prints FAILS.

```pascal
program t5p8p3p10dl(output);
var
  r1:record
      a,b:integer
    end;
  r2:record
      c,d:integer
    end;
  r3:record
      e,f:integer
    end;
  counter:integer;
begin
  counter :=0;
  with r1 do
    a:=5;
  with r1,r2,r3 do
  begin
    c:=a;
    e:=a;
  end;
  if c<>5 then
    counter:=counter+1;
  if r2.c<>5 then
    counter:=counter+1;
  if counter=2 then
    writeln(' PASS 6.8.3.10-1, WITH');
  else
    writeln(' FAIL 6.8.3.10-1, WITH');
end.
```

This test checks that a field identifier is correctly identified when a with statement is invoked. The compiler fails if the program does not compile or the program prints FAILS.

```pascal
program t5p8p3p10d2(output);
var
  r:record
    i,j:integer
  end;
  i: integer;
begin
  i:=10;
  with r do
  begin
    i:=5;
    if (i=10) and (r.i=5) then
      writeln(' PASS 6.8.3.10-2, WITH')
    else
      writeln(' FAIL 6.8.3.10-2, WITH');
  end.
```

[TEST 6.8.3.10-1, CLASS=CONFORMANCE]
program t5p8p3p10d3(output);
var
  r1: record
    i,j,k: integer
  end;
  r2: record
    i,j: integer
  end;
  r3: record
    i: integer
  end;
with r1 do
  begin
    i:=0;
    j:=0;
    k:=0
  end;
with r2 do
  begin
    i:=0;
    j:=0
  end;
with r3 do
  begin
    i:=0;
    with r1,r2,r3 do
    begin
      i:=5;
      j:=6;
      k:=7
    end;
if (r1.i=0) and (r1.j=0) and (r2.i=0) and (r1.k=7) and (r2.j=5) and (r3.i=5) then
  writeln(' PASS...6.8.3.10-3, WITH EVALUATION');
else
  writeln(' FAIL...6.8.3.10-3, WITH EVALUATION');
end.
This test checks that the selection of a variable in the record-variable-list is performed before the component statement is executed. The compiler fails if the program does not compile or the program prints FAIL.

```pascal
program t6p8p3p10d5(output);
type
  pointer = Trecordtype;
  recordtype = record
    data: integer;
    link: pointer
  end;
var
  counter: integer;
  p, q, r: pointer;
begin
  counter := 0;
  new(p);
  pT.data := 0;
  new(q);
  qT.data := 1;
  qT.link := nil;
  pT.link := q;
  new(r);
  rT.data := 0;
  rT.link := nil;
  qT.link := r;
  with pT, linkT, linkT do
    begin
      i := 5;
      end;
  if ((rT.i = 5) and (qT.data = 1) and (pT.data = 0)) then
    writeln('PASS..6.8.3.10-6, WITH');
  end;
end.
```

This test checks that the order of evaluation of the record-variable-list in a with statement is correctly implemented. The compiler fails if the program prints FAIL.

```pascal
program t6p8p3p10d6(output);
type
  pp = Tptr;
  ptr = record
    i: integer;
    link: pp
  end;
var
  p, q, r: pp;
begin
  new(p);
  pT.i := 0;
  new(q);
  qT.i := 0;
  pT.link := q;
  new(r);
  rT.i := 0;
  qT.link := r;
  with pT, linkT, linkT do
    begin
      i := 5;
      if (rT.i = 5) and (qT.i = 0) and (pT.i = 0) then
        writeln('PASS..6.8.3.10-6, WITH')
      else
        writeln('FAIL..6.8.3.10-6, WITH');
    end.
end.
```
This test checks that with statements may be nested to 15 levels. The test may break a compiler limit in some compilers, particularly if a register is allocated for every selected variable.

program t6p8p3p10d7(output);
type
rec1 = record
  i: integer
end;
rec2 = record
  i: integer
end;
rec3 = record
  i: integer
end;
rec4 = record
  i: integer
end;
rec5 = record
  i: integer
end;
rec6 = record
  i: integer
end;
rec7 = record
  i: integer
end;
rec8 = record
  i: integer
end;
rec9 = record
  i: integer
end;
rec10 = record
  i: integer
end;
rec11 = record
  i: integer
end;
rec12 = record
  i: integer
end;
rec13 = record
  i: integer
end;
rec14 = record
  i: integer
end;
rec15 = record
  i: integer
end;
var
  p8 = rec8;
p9 = rec9;
p10 = rec10;
p11 = rec11;
p12 = rec12;
p13 = rec13;
p14 = rec14;
p15 = rec15;
begin
  writeln(' >15 LEVELS OF WITH STATEMENTS ALLOWED...6.8.3.10-7');
  end.
This program checks that the functions `eoln` and `eof` are correctly implemented. The compiler fails if the program does not compile or the program prints `FAIL`.

```pascal
program t5p9pl1 (f, output);
var
  f: text;
  counter: integer;
  c: char;
begin
  rewrite (f);
  counter := 0;
  writeln(f, '1 2 3 4 5 f');
  reset (f);
  read (f, a, b, c, d, e);
  reset (f);
  al := fT; get(f);
  bl := fT; get(f);
  if (a = al) and (b = bl) then
    writeln('PASS...6.9.2-2, READ');
  else
    writeln('FAIL...6.9.2-2, READ');
end.
```

This test checks that a single read statement with many variables is equivalent to many read statements containing one variable each. The compiler fails if the program does not compile or the program prints `FAIL`.

```pascal
program t5p9p2d1 (f, output);
var
  f: text;
  a, b, c, d, e: integer;
  al, bl, cl, dl, el: integer;
begin
  rewrite (f);
  writeln(f, 'A');
  reset (f);
  read (f, a, b, c, d, e);
  reset (f);
  if (a = al) and (b = bl) and (c = cl) and (d = dl) and (e = el) then
    writeln('PASS...5.9.2-1, READ');
  else
    writeln('FAIL...5.9.2-1, READ');
end.
```

This test checks that a read of a character variable is equivalent to correctly positioning the buffer variable. The compiler fails if the program does not compile or the program prints `FAIL`.

```pascal
program t5p9p2d2 (f, output);
var
  f: text;
  a, b, al, bl: char;
begin
  rewrite (f);
  writeln(f, ' ABC');
  reset (f);
  read (f, a);
  reset (f);
  a := fT; get(f);
  bl := fT; get(f);
  if (a = al) and (b = bl) then
    writeln('PASS...6.9.2-2, READ');
  else
    writeln('FAIL...6.9.2-2, READ');
end.
```
This test checks that integers and reals are read correctly from a file. The compiler fails if the program does not compile or the program prints FAIL.

program t6p9p2d3(f,output);
begin
  var
    f:text;
    i,j:integer;
    r,s:real;
  begin
    rewrite(f);
    writeln(f,' 123 123.456 5 123Eh');
    reset(f);
    read(f,i,r,j,s);
    if (i=123) and (r=123.456) and (j=5) and (s=123E6) then
      writeln(' PASS ...6.9.2-3, READ')
    else
      begin
        if (i=123) and (j=5) then
          writeln(' FAIL ...6.9.2-3, READ
REAL CONVERSIONS')
        else
          writeln(' FAIL ...6.9.2-3, READ')
      end;
  end.

This test checks that an error is produced when an attempt is made to read an integer but the sequence of characters on the input file does not form a valid signed integer.

program t6p9p2d4(f,output);
begin
  var
    f:text;
    i:integer;
  begin
    rewrite(f);
    writeln(f,'ABC123');
    reset(f);
    read(f,i);
    if (i='D') then counter:=counter+1;
    if (counter<3) then
      writeln(' ERROR NOT DETECTED...6.9.2-4');
    end.

This test checks that an error is produced when an attempt is made to read a real but the sequence of characters on the input file does not form a valid real.

program t6p9p2d5(f,output);
begin
  var
    f:text;
    r:real;
  begin
    rewrite(f);
    writeln(f,'ABC123.456');
    reset(f);
    read(f,r); {should cause an error!}
    writeln(' ERROR NOT DETECTED...6.9.2-5');
  end.

This test checks that readln is correctly implemented. The compiler fails if the program prints FAIL.

program t6p9p3dl(output);
begin
  counter:=0;
  rewrite(f);
  writeln(f,'ABC');
  writeln(f,'DE');
  reset(f);
  readln(f,a,b,c);
  if (a='D') then counter:=counter+1;
  reset(f);
  readln(f,a,b,c);
  if (counter<3) then
    writeln(' ERROR NOT DETECTED...6.9.3-1');
    writeln(' PASS ...6.9.3-1, READLN');
else
  writeln(' FAIL ...6.9.3-1, READLN');
end.
This test checks that a write procedure with many parameters is equivalent to many write procedures with one parameter each. The compiler fails if the program does not compile or the program prints FAIL.

program t6p9p4d1(f, output);
var
  f: text;
  a, b, c, d, e: char;
  al, b1, c1, d1, e1: char;
  counter: integer;
begin
  counter := 0;
  rewrite(f);
  a := 'A';
  b := 'B';
  c := 'C';
  d := 'D';
  e := 'E';
  write(f, a, b, c, d, e);
  writeln(f);
  reset(f);
  read(f, al, b1, c1, d1, e1);
  if (a = al) and (b = b1) and (c = c1) and (d = d1) and (e = e1) then
    counter := counter + 1;
  writeln(f);
  reset(f);
  rewrite(f);
  write(f, a, b);
  writeln(f, a, b);
  reset(f);
  read(f, a, b);
  if (a = a) and (b = b) then
    writeln(' PASS...6.9.4-1, WRITE')
  else
    writeln(' FAIL...6.9.4-1, WRITE');
end.

This test checks that the default value for the field width of a character type is one. The compiler fails if the program does not compile or the program prints FAIL.

program t6p9p4d2(f, output);
var
  f: text;
  a, b: char;
begin
  rewrite(f);
  a := 'A';
  b := 'B';
  writeln(f, a, b);
  reset(f);
  read(f, a, b);
  if (a = 'A') and (b = 'B') then
    writeln(' PASS...6.9.4-2, WRITE')
  else
    writeln(' FAIL...6.9.4-2, WRITE');
end.

This test checks the implementation of integer output. The compiler fails if the program does not compile or the program prints FAIL.

program t6p9p4d3(f, output);
var
  f: text;
  a: char;
  b: packed array [1..26] of char;
  i: integer;
begin
  rewrite(f);
  a := 'A';
  b := 'B';
  writeln(f, a, b);
  reset(f);
  for i := 1 to 26 do
    read(f, b[i]);
  if (b[0] = 0) and (b[1] = 99100) and (b[1111] = ') then
    writeln(' PASS...6.9.4-3, WRITE INTEGERS')
  else
    writeln(' FAIL...6.9.4-3, WRITE INTEGERS');
end.
This program checks that real numbers are correctly written to text files. The compiler fails if the program does not compile or the program prints FAIL.

program testp9p4d4(f,output);
var
  f:text;
  a:packed array [1..26] of char;
  b:packed array [1..24] of char;
  i:integer;
  counter:integer;
begin
  rewrite(f);
  counter:=0;
  writeln(f,0.0:6,1.0:6,1.0:10);
  reset(f);
  for i:=1 to 26 do
    read(f,a[i]);
  if (a='0.0 1.0 1.000E+00') then
    counter:=counter+1;
  rewrite(f);
  writeln(f,0.0:4:1,1.0:6:1,-1.0:6:1,123.456:7:3);
  reset(f);
  for i:=1 to 24 do
    read(f,b[i]);
  if (b='0.0 1.0 -1.0 123.456') then
    counter:=counter+1;
  if (counter=2) then
    writeln(' PASS...6.9.4-4, WRITE REALS');
  else
    writeln(' FAIL...6.9.4-4, WRITE REALS');
end.

This program determines the implementation defines value which represents the number of digit characters written in an exponent.

program testp9p4d5(f,output);
var
  f:text;
  c:char;
  i:integer;
begin
  rewrite(f);
  writeln(f,1.0:10,'ABC');
  reset(f);
  repeat
    read(f,c);
    until (c='E');
  read(f,c);
  i:=i+1;
  repeat
    read(f,c);
    i:=i+1;
    until (c='A');
  writeln(' THE NUMBER OF DIGITS WRITTEN IN AN EXPONENT IS',i:5);
end.
This test checks that strings are correctly written onto a text file. The compiler fails if the program does not compile or the program prints FAIL.

```pascal
program t6p9p4d6(f,output);
var
  f:text;
  i,j,k,counter:integer;
  c:char;
begin
  rewrite(f);
  counter:=0;
  for i := 1 to 10 do
    writeln(f,'AAAAA':i,' B':1);
  writeln(f,'BBBBB','C':1);
  reset(f);
  for i:=1 to 10 do
    begin
      for j:=6 to i do
        begin
          read(f,c);
          if (c=' ') then counter :=counter+ 1;
        end;
      if (i>5) then k:=5 else k:=i;
      for j :=1 to k do
        begin
          read(f,c);
          if(c='A') then counter :=counter+ 1;
        end;
      read(f,c);
      if (c='B') then counter :=counter+ 1;
    end;
  for i:=1 to 5 do
    begin
      read(f,c);
      if (c='B') then counter:=counter+ 1;
    end;
  readln( f);
end.
```

This test checks that boolean variables are correctly written to text files. The compiler fails if the program does not compile or the program prints FAIL.

```pascal
program t6p9p4d7(f,output);
var
  f:text;
  b,c :boolean;
  a:packed array[1..10] of char;
  i: integer;
begin
  rewrite(f);
  b:=true;
  c:=not b;
  writeln(f,b:5,c:5);
  reset(f);
  for i :=1 to 10 do
    read(f,a[i]);
  if (a='TRUE FALSE') then
    writeln(' PASS...6.9.4-7, WRITE BOOLEAN')
  else
    writeln(' FAIL...6.9.4-7, WRITE BOOLEAN');
end.
```

This test attempts to output integers whose field width parameter are zero or negative. The compiler deviates if the program prints DEVIATES.

```pascal
program t6p9p4d9(output);
var
  i:integer;
begin
  for i:=10 downto -1 do
    writeln(' FAIL...6.9.4-6, WRITE STRINGS')
end.
```

This test attempts to output an integer number using a real format. The compiler deviates if the program prints DEVIATES.

```pascal
program t6p9p4d8(output);
var
  i:integer;
begin
  i:=123;
  writeln(' FAIL...6.9.4-8, WRITE');
end.
```

This test attempts to output integers whose field width parameter are zero or negative. The compiler deviates if the program prints DEVIATES.

```pascal
program t6p9p4d9(output);
var
  i:integer;
begin
  for i:=10 downto -1 do
    writeln(' PASS...6.9.4-9, WRITE')
end.
```
This program checks that data written appears on the output file regardless of the omission of a line marker. The common error is to buffer output and fail to flush the buffers at end of job.

```pascal
program t5p9p4dl0(output);
begin
  write(' OUTPUT IS FLUSHED AT END_OF_JOB...

program t5p9p4dl1(output);
var
  f:text;
  c:char;
  i,j:integer;
  function readfield:integer;
  var
    i: integer;
  begin
    i:=0;
    repeat
      read(f,c);
      i :=i+1;
    until (c='Z');
    readfield:=i-1;
   end;
begin
  rewrite(f);
  writeln(f,'Z',100,'Z');
  writeln(f,false,'Z',true,'Z');
  writeln(f,1.0,'Z',1000.0,'Z');
  reset(f);
  writeln(' IMPLEMENTATION DEFINED DEFAULT FIELD WIDTH VALUES');
  i:=readfield;
  j:=readfield;
  if (i=j) then
    writeln(' INTEGERS:',i:5,' CHARACTERS')
  else
    writeln(' THE VALUE VARIES ACCORDING TO THE SIZE OF THE INTEGER');
  readln(f);
  i:=readfield;
  j:=readfield;
  if (i=j) then
    writeln(' BOOLEAN:',i:5,' CHARACTERS')
  else
    writeln(' THE VALUE VARIES ACCORDING TO THE BOOLEAN VALUE');
  readln(f);
  i:=readfield;
  j:=readfield;
  if (i=j) then
    writeln(' REAL:',i:5,' CHARACTERS')
  else
    writeln(' THE VALUE VARIES ACCORDING TO THE SIZE OF THE REAL');
end.
```
This program checks whether an unpacked array of characters can be output. The compiler deviates if the program prints DEVIATES.

program t6p9p4d12(output);
var
s:array[1..3] of char;
begins[1]:=R'; s[2]:=A'; s[3]:='N';
writeln(' RAN=' s);
writeln(' DEVIATES ...6.9.4-12, WRITE');
end.

This program attempts to perform recursive I/O using a different file for the second I/O action.

program t6p9p4d13(f,output);
var
f :text;
function a(i:integer):integer;
begina:=i;
writeln(f, i) ;
a:=i;
end;
begin
write(f, i);    writeln(f) ;
 resets(f);
for i:=1 to 10 do
  read(f,a[i]) ;
 resets(f);
for i:=1 to 10 do
  read(f,b[i]);
if(a=b) then
  writeln(' PASS ...6.9.5-1, WRITELN');
else
  writeln(' FAIL ...6.9.5-1, WRITELN');
end.

This program attempts to perform recursive I/O using the same file for the second I/O action. The semantics of write are not sufficiently well-defined to establish what should occur. It depends on evaluation orders, etc., which is why this test is in the quality section.

program t6p9p4d14(f,output);
function a(i:integer):integer;
begina:=i;
writeln(a());
a:=i;
end;
begin
writeln(a());
 writeln('RECURSIVE I/O ALLOWED USING THE SAME FILE ...6.9.4-14');
end.
This program checks that the procedure page is implemented. This conformance test is unable to determine whether the compiler passes or fails - the user must check that a page has been generated.

```pascal
program t6p9p6dl(output);
begin
  writeln(' PAGE GENERATION TEST');
  page(output);
  writeln(' IF THIS LINE IS PRINTED ON THE TOP OF A NEW PAGE');
  writeln(' THEN PASS ••• 6.9.6-1, PAGE');
  writeln(' ELSE FAIL ••• 6.9.6-1, PAGE');
end.
```

This test checks the effect of using a default file not declared in the program heading. The compiler deviates if the program prints DEVIATES.

```pascal
program t6pl0dl(input);
begin
  writeln(' DEVIATES ••• 6.10-1, FILE DECLARATION');
end.
```

This program checks the effect of doing a rewrite on the standard file output. The effect is implementation dependent.

```pascal
program t6p10d2(output);
begin
  rewrite(output);
  writeln(' IMPLEMENTATION DEPENDENT ••• 6.10-2');
  writeln(' A REWRITE HAS BEEN PERFORMED ON FILE OUTPUT');
end.
```

This program checks that the default file output is implicitly declared at the program level by attempting to redefine it. The file input should be identical, of course. The test should not compile.

```pascal
program t6p10d3(output);
var
  output: integer;
begin
  output:=1;
  writeln(' DEVIATES ••• 6.10-3, OUTPUT REDEFINED');
end.
```

This program has no program statement. Some compilers may assume the existence of such a statement if none is present. The compiler deviates if the program compiles and prints DEVIATES.

```pascal
var
  i: integer;
begin
  i:=5;
  writeln(' DEVIATES ••• 6.10-4, PROGRAM');
end.
```

This test checks whether equivalent symbols can be used for the standard reference representation. The equivalent symbols for comment delimiters are tested. They are implemented if the program prints ALTERNATE COMMENT DELIMITERS IMPLEMENTED.

```pascal
program t5p1ldl(output);
(* Test of alternate comment delimiters *)
begin
  (* test of alternate comment delimiters. If these delimiters
   are not implemented the compiler will give a syntax error. *)
  writeln(' ALTERNATE COMMENT DELIMITERS IMPLEMENTED ••• 6.11-1');
end.
```
THREE SAMPLE VALIDATION REPORTS

A fourth report came to hand very late in the preparation of this issue, and it is also included.

Introduction

In this section we present three samples of Validation Reports on processors. Care is needed in the interpretation of these Reports for several reasons. Firstly, they are a snapshot in time of the processor concerned; some of the reported flaws will be fixed by the maintainers, perhaps even before this is printed. Secondly, some processors contain intended extensions, or have an interpretation which anticipates a change in the draft Standard in its route to finalization. Nevertheless, it is felt that publication of these Reports will

(a) encourage other users to test processors accessible to them and publish the results in Pascal News,
(b) indicate likely portability problem areas, and
(c) illustrate the type of report which will be meaningful to users of Pascal.

It must be emphasized that these reports are simply of processors which were reasonably convenient for us to test. The report on the Burroughs B6700 compiler originating at the University of California is a user's viewpoint of an unmodified system. There is no good reason to suspect that the report is particularly bad (or particularly good). It is likely to be representative of the results other users will achieve with their processors.

The report on the Burroughs B6700/7700 compiler originating at the University of Tasmania is somewhat different. Firstly, it is our own processor, and the report is therefore prepared with greater knowledge of what is happening (not simply noting a bald failure for unknown reasons as we have had to record in other situations). Secondly, and more importantly, it has been the prime testbed for the validation suite which has resulted in most of the minor faults being fixed as soon as they are detected. Consequently, only a few of the more difficult areas remain to be reported by the Validation Suite.

The report of the P4 compiler gives an indication of how the portable Pascal systems conform to the standard.

The tests which led to these reports were carried out by:

R.A. Freak
C.D. Keen

Annotation and analysis were carried out by:

R.A. Freak
A.H.J. Sale
C.D. Keen

(For the benefit of Pascal users who want to carry out similar Validation Tests on their processors, we estimate the time required to do this as about 1-5 man days, depending on familiarity with the processor, turn-around time, etc. Fixing the flaws, of course, takes a lot longer).
Pascal Validation Suite Report

Pascal Processor Identification
Computer: Burroughs B6700
Processor: B6700 Pascal version 2.9.178.008 (University of California at San Diego compiler)

Test Conditions
Tester: R.A. Freak (a user at the University of Tasmania)
Date: August 1979
Validation Suite Version: 2.2

Conformance Tests
Number of tests passed: 118
Number of tests failed: 21 (13 basic causes)

Details of failed tests:
Test 6.1.5-2 failed because long numbers are not accepted by the compiler.
Tests 6.4.3.2-3, 6.6.3.1-2 and 6.8.2.1-1 failed because booleans are not permitted to be used as array indexes.
Tests 6.4.3.3-1 and 6.4.3.3-3 fail because empty records or empty fields are not permitted by the compiler.
Test 6.4.3.5-1 shows that a file of pointer is not allowed.
Tests 6.4.3.5-2, 6.6.5.2-3, 6.6.5.2-4, 6.6.5.2-5, 6.9.1-1 all fail because the implementation of textfiles is non-standard - particularly the handling of eof and eoln.
Test 6.4.3.5-4 indicates that the file output is not flushed at the end of a program.
Test 6.5.1-1 shows that a file of char is not permitted in a record.
Test 6.6.3.1-5 failed because of an inconsistency in the implementation of procedure parameters.
Test 6.6.5.4-1 failed because the procedures pack and unpack have not been implemented according to the standard.

Deviance Tests
Number of deviations correctly detected: 62
Number of tests showing true extensions: 5
Number of tests not detecting erroneous deviations: 28 (11 basic causes)

Details of extensions:
Test 6.1.5-6 shows that a lower case e may be used in real numbers (e.g. 1.602E-20).
Test 6.1.7-6 shows that strings can have bounds other than 1..n.
Test 6.1.7-11 shows that a null string is accepted by the compiler.
Tests 6.8.3.9-9 and 6.8.3.9-14 indicate that a non-local variable or a global variable may be used for a for statement control variable.

Details of deviations not detected:
Test 6.1.2-1 shows that the reserved word nil may be redefined.
Tests 6.2.2.4, 6.2.2.9, 6.3.5, 6.3-6, 6.4.1-3 contain a scope error which is not detected by the compiler.
Tests 6.4.5-2, 6.4.5-3, 6.4.5-4, 6.4.5-5 and 6.4.5-13 indicate that type compatibility is used with var parameters rather than enforcing identical types.
Test 6.6.1-6 shows that a forward procedure declaration without a procedure body is not detected.
Test 6.6.1-5 shows that a function without an assignment to the function variable in its block compiles and runs.
Tests 6.6.3.2-2, 6.6.3.6-2 and 6.6.3.6-4 show that function parameters are assignment compatible.
Test 6.6.5-3 shows that the function \texttt{ddi} may be used with a real parameter.

Test 6.7.2.2-9 shows that a unary plus sign may be used with character operands.

Tests 6.8.2.4-2, 6.8.2.4-3 and 6.8.2.4-4 show that a \	exttt{goto} between branches of a statement is permitted.

Tests 6.8.3.8-2, 6.8.3.8-3, 6.8.3.8-4, 6.8.3.8-5, 6.8.3.8-16 and 6.8.3.8-19 show that a for statement control variable may be altered during the execution of the for loop.

Test 6.9.4-9 indicates that a negative field width may be used in a write statement.

Error handling

Number of errors correctly detected: 18

Number of errors not detected: 27 (12 basic causes)

Details of errors not detected

Tests 6.4.3.3-5, 6.4.3.3-6, 6.4.3.3-7 and 6.4.3.3-8 indicate that no checking is performed on the tag field of variant records.

Test 6.4.3.3-12 shows that an assignment to an empty record is not detected.

Test 6.4.6-4 indicates that no bounds checking is performed on subranges.

Tests 6.4.6-7 and 6.4.6-8 indicate that no bounds checking is performed on set operations.

Test 6.6.2-6 shows that the use of a function without an assignment to the function-value-variable is permitted.

Tests 6.6.5.2-1, 6.6.5.2-2, 6.6.5.2-6 and 6.6.5.2-7 fail because \texttt{I/O} has not been implemented strictly according to the standard - particularly the handling of \texttt{eof} and \texttt{eoln}.

Tests 6.6.5.3-3, 6.6.5.3-5 and 6.6.5.3-6 fail because no check is performed on the pointer parameter of \texttt{dispose}.

Tests 6.6.5.3-7, 6.6.5.3-8 and 6.6.5.3-9 fail because no checks are inserted to check pointers after they have been assigned a value using the variant form of new.

Tests 6.6.6.4-4, 6.6.6.4-5 and 6.6.6.4-7 fail because no bounds checks are inserted for the \texttt{eaa}, \texttt{prec} or \texttt{chr} functions.

Tests 6.7.2.2-6 and 6.7.2.2-7 fail because integer overflow/underflow is not detected.

Test 6.7.2.4-1 fails because operations on overlapping sets are not detected.

Tests 6.8.3.9-6 and 6.8.3.9-17 fail because a for control variable is not invalid after the execution of the for loop.

Implementation defined

Number of tests run: 15

Number of tests incorrectly handled: 1

Details of implementation-dependence

Test 6.4.2.2-7 shows \texttt{maxint} to be 549755813887.

Tests 6.4.3.4-2 and 6.4.3.4-4 show that all set bounds must be positive. A set of char is permitted.

Test 6.6.6.1-1 shows that no standard functions may be used as parameters.

Test 6.6.6.2-11 details some machine characteristics regarding number formats.

Tests 6.7.2.3-2 and 6.7.2.3-3 show that boolean expressions are fully evaluated.

Tests 6.8.2.2-1 and 6.8.2.2-2 show that a variable is selected before the expression is evaluated in an assignment statement.

Test 6.9.4-11 details the default field width specifications: 10 for integers and booleans. The output format for reals fails in test 6.9.4-5.

Test 6.10-2 indicates that a rewrite on the standard file output is permissible.

Tests 6.11-1, 6.11-2 and 6.11-3 show that the alternative comment delimiters have been implemented, as have the alternative pointer symbols. No other equivalent symbols have been implemented.

Quality Measurement

Number of tests run: 23

Number of tests incorrectly handled: 0

Results of tests:

Test 5.2.2-1 shows that identifiers are not distinguished over their whole length; only the first 12 characters are used.
Test 6.1.3-3 shows the number of significant characters in an identifier to be 12.

Test 6.1.8-4 shows that no warning is given if a valid statement or a semicolon is detected in a comment.

Tests 6.2.1-8, 6.2.1-9 and 6.5.1-2 indicate that large lists of declarations may be made in each block.

An array with an integer index type is not permitted (test 6.4.3.2-4).

Test 6.4.3.3-9 shows that variant fields of a record occupy the same space, using the declared order.

Test 6.4.3.4-5 (Warshall's algorithm) took 10.497744 seconds CPU time and 169 bytes on the Burroughs B6700.

Test 6.6.1-7 shows that procedures cannot be nested to a level greater than 8.

Tests 6.6.6.2-6, 6.6.6.2-7, 6.6.6.2-8, 6.6.6.2-9 and 6.6.6.2-10 tested the sqrt, tan, exp, sin/cos and ln functions and all tests were completed successfully, without any significant errors in the values.

Test 6.7.2.2-4 shows that div and mod have been implemented consistently. mod returns the remainder of div.

Test 6.8.3.5-2 shows that case constants do not have to be of the same type as the case-index, if the case-index is a subrange, but the constants must be compatible with the case-index.

Test 6.8.3.5-8 shows that a large case statement (>256 selections) is permissible.

Test 6.8.3.9-18 indicates that range checking is always used in a case statement after a for statement to check the for variable.

Tests 6.8.3.9-20 and 6.8.3.10-7 indicate that for and with statements may be nested to a depth greater than 15.

Test 6.9.4-10 shows that file buffers are flushed at the end of the program.

Test 6.9.4-14 indicates that recursive I/O is permitted, using the same file.
B6700-Tas
PASCAL VALIDATION SUITE REPORT

Pascal Processor Identification

Computer: Burroughs B6700
Processor: B6700 Pascal version 2.9.001
(University of Tasmania compiler)

Test Conditions

Tester: R.A. Freak (implementation/maintenance team member)
Date: August 1979

Validation Suite Version: 2.2

Conformance Tests

Number of tests passed: 117
Number of tests failed: 22

Details of failed tests:
Most of the failed tests fall into two categories - the B6700 Pascal I/O is non-standard and the passing of procedure/function parameters has not been implemented.

Tests 6.4.3.3-1, 6.4.3.5-1, 6.4.3.5-2, 6.4.3.5-3, 6.5.1-1, 6.5.3.4-1, 6.6.5.2-3, 6.6.5.2-4, 6.6.5.2-5, 6.9.1-1, 6.9.2-2, 6.9.3-1, 6.9.4-3, 6.9.4-4, 6.9.4-7, 6.9.4-15, all fail because text has not been predefined, or the eof action or output format is not strictly standard-conforming.

Tests 6.6.3.1-3, 6.6.3.1-5, 6.6.3.4-1, 6.6.3.4-2 and 6.6.3.5-1 fail because the passing of procedure/function parameters has not yet been implemented.

Test 6.4.3.3-1 fails because an empty record containing a semi-colon produces a syntax error.

Details of extensions:
Test 6.1.5-6 shows that the lower case e may be used in real numbers (for example 1.602e-20).
Test 6.6.2-5 shows that a function without an assignment to the function variable in its block compiles - the error is detected at run time as an uninitialized value.
Test 6.9.3-8 shows that integers may be written using real formats.
Test 6.10-1 shows that the file parameters in the program heading are ignored in B6700 Pascal.
Test 6.10-3 shows that the file output may be redefined at the program level.

Details of deviations not detected:
Test 6.1.2-1 shows that nil may be redefined.
Tests 6.2.2-4, 6.3-6 and 6.4.1-3 show that a common scope error was not detected by the compiler.
Tests 6.4.5-2, 6.4.5-3, 6.4.5-4, 6.4.5-5 and 6.4.5-13 indicate that type compatibility is used with var parameter elements rather than enforcing identical types.

Tests 6.8.2.4-2, 6.8.2.4-3 and 6.8.2.4-4 show that a goto between branches of a statement is permitted.
Test 6.9.4-9 shows that integers may be written even though the field width is too small, but the format used is non-standard.

Details of failed tests:
Tests 6.6.3.5-2, 6.6.3.6-2, 6.6.3.6-3, 6.6.3.6-4 and 6.6.3.6-5 fail because procedure/function parameters have not been implemented.
Test 6.9.4-15 fails because text has not been defined.

Error Handling

Number of errors correctly detected: 22
Number of errors not detected: 23 (6 basic causes)

Details of errors not detected: The errors not detected fall into a number of categories:
Tests 6.4.3.3-5, 6.4.3.3-6, 6.4.3.3-7 and 6.4.3.3-8 indicate that no checking is performed on the tag field of variant records.
An assignment to an empty record is not detected in test 6.4.3.3-12.
Tests 6.4.6-4, 6.4.6-5, 6.4.6-7, 6.4.6-8, 6.5.3.2-1, 6.6.6.4-7 and 6.7.2.4-1 indicate that no bounds checking is performed on array subscripts, subranges, set operations or the CHR function.

Tests 6.6.5.2-1, 6.6.5.2-2, 6.6.5.2-6 and 6.6.5.2-7 fail because I/O has not been implemented strictly according to the standard.

Tests 6.6.5.3-3, 6.6.5.3-4, 6.6.5.3-5 and 6.6.5.3-6 fail because dispose always returns a nil pointer in B6700 Pascal and no check is performed on the pointer parameter.

Tests 6.6.5.3-7, 6.6.5.3-8 and 6.6.5.3-9 fail because no checks are inserted to check pointers after they have been assigned a value using the variant form of new.

Implementation defined

Number of tests run: 15
Number of tests incorrectly handled: 1

Details of implementation dependence:

Test 6.4.2.2-7 shows maxint to be 549755813887.
Test 6.4.3.4-2 and 6.4.3.4-4 show that the set bounds are 0 and 47. A set of char is not permitted.
Test 6.6.6.2-11 details some machine characteristics regarding number formats.
Tests 6.7.2.3-2 and 6.7.2.3-3 show that boolean expressions are fully evaluated.
Tests 6.8.2.2-1 and 6.8.2.2-2 show that a variable is selected before the expression is evaluated in an assignment statement.
Tests 6.9.4-5 and 6.9.4-11 show that the default size for an exponent field on output is 2; for a real number it is 15 and the size varies for integers and booleans according to the value being written.
Test 6.10-2 indicates that a rewrite on the standard file output is permissible.
Tests 6.11-1, 6.11-2 and 6.11-3 show that the alternative comment delimiters have been implemented, as have the alternative pointer symbols. No other equivalent symbols have been implemented.
Test 6.6.6.1-1 fails because function parameters have not been implemented, and therefore standard functions cannot be used as procedure/function parameters.

Test 6.6.6.2-6, 6.6.6.2-7, 6.6.6.2-8, 6.6.6.2-9, and 6.6.6.2-10 tested the sqrt, atan, exp, sin/cos and ln functions and all tests were completed successfully, without any significant errors in the values.

Test 6.7.2.2-4 shows that div has been implemented consistently for negative operands, returning trunc. mod returns the remainder of div.
Tests 6.8.3.5-2 shows that case constants do not have to be of the same type as the case-index, if the case-index is a subrange, but the constants must be compatible with the case-index.
Test 6.8.3.5-8 shows that a large case statement (>256 selections) is permissible.
Test 6.8.3.9-18 indicates that range checking is always used in a case statement after a for statement to check the for variable.
Test 6.9.4-10 shows that file buffers are flushed at the end of a block but test 6.9.3-14 indicates that recursive I/O using the same file may produce unexpected results.
Extensions

Number of tests run = 1

Test 6.8.3.5-14 shows that the otherwise clause in a case statement has been implemented according to the accepted convention.

B6700 Pascal - Future Plans and Commentary on Results

The Validation Suite has shown up a number of flaws in the Tasmania B6700 compiler, as documented in the preceding report. We expect that other compilers will typically fare worse in the number of different flaws detected because we have had the benefit of experience (and fixing bugs) as we were developing the suite. This brief document outlines what we expect to do about them.

1. Minor Flaws

Some of the reported flaws are easy to fix, and have survived to be reported to you only by an oversight, or because the relevant test has only recently been added to the suite. Examples are semicolons in empty records, and incorrect var parameter typing. These will be fixed as soon as possible, and probably before this document is released.

2. Substantial Flaws

Two major flaws have survived because they require a reasonable amount of work to repair. These are the deviations of the i/o system, which seem to indicate a revision of the i/o run-time system, and procedure and function parameters which could not have been implemented until the draft standard solution was published due to the insecurities in the original version. These are under revision, and will be fixed shortly. Procedure and function parameters particularly should not take long.

3. Long-term and medium-term improvements

In the long term, we plan to implement techniques which we have evolved or borrowed for improving the security of Pascal in our compiler, such as checking bounds efficiently outside the B6700 hardware checks, providing correct scoping checks, checking the validity of goto-statements, etc.

August 1979
R.A. Freak & A.H.J. Sale

PDP-11 OMSI 1

VALIDATION REPORT

MACHINE: DEC PDP-11 running RSX11M-03
COMPILER: OMSI Pascal-1 (Field Test Version X1.2)
DATE: 1979 September 9 & 10
TESTS BY: Barry Smith, Oregon Software
(Implementation/maintenance team)
ANOTATED BY: A.H.J. Sale (1979 September 13)

Copyright A.H.J. Sale 1979
Not to be distributed nor reproduced without permission.

CONFORMANCE TESTS

Number of tests attempted: 131
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.

6.2.3-3 Pointer scope is not handled correctly, so that correct programs fail to compile.

6.2.3-8 Assignment to function-identifier from within nested procedure or function generates bad code.

6.4.8.31 Empty record types with semicolons and empty case variants are not permitted.

6.6.3.5-6 and -9 An unknown interaction between RSX I/O on temporary files and the implementation of the run-time support.

6.6.5.3-8 Equal compatible sets of different basetypes do not compare equal. Pascal-I 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 'J'..'~' (64 chars),)

6.6.6.3-6 and 6.6.6.4-2 Only J&W procedural parameters allowed, not the N462 versions. The second test is relevant to the feature actually implemented, but has not been run with modifications.

6.6.6.8-9 Does not check eof on an empty temporary file.
Pack and unpack not implemented.

Empty field specifications not allowed in record declarations.

Main and eof not correct: relation between EOLS and implementation
causes unknown fault.

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

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

**ERROR HANDLING TESTS**

Number of tests attempted: 25
Number of extensions: 2 (as stated by B. Smith)
Number of deviations: 41 (25 causes)

- Allows real number constants without digits after point.
- Packed is ignored so that packed array of char is
  identical to array of char, and similarly with other structures.
- The requirements to be a string-type are not checked, allowing
  default programs to execute.
- The requirements to be a string-type are not checked, together with
  an obvious error, allows erroneous values to be given to a type.
- Allows empty string: le’ is equivalent to packed array[1..0]
  of char.
- Incorrect scope allows incorrect program to compile.
- Invalid program executes with (a) function whose identifier is
  inaccessible and (b) another function has an attempted assignment outside
  its block.
- Function-identifier may be assigned to outside its block.
- Signed characters, strings and enumerated types are allowed.
- Allows packed scalars, subranges, is not restricted to structures.
- "packed" type-identifier.
- String types are allowed to have non-integer subrange indexes.
- Set of real erroneously not detected.
- Var parameters which are compatible but not identical are allowed.
- Non-identical array types allowed as var parameters.
- Non-identical pointer types allowed as var parameters.
- Compiles file assignment as descriptor copy, and
  similarly for records containing file components.
- Allows function definitions without any assignment to function-
  identifier.
- Allows goto statements to transfer into structured
  statement components.
- Any assignment to a for-control-variable is
  allowed inside the controlled statement, and it in fact changes the value.
- Allows a for-control-variable to be program-global,
  non-local, or a var parameter.
- Two loops using some variable interact to produce infinite
  loop construction, and other insecurities.
- I I lore program parameters, allowing use of external file not stated.
- The files input and output are not implicitly declared at the
  program level, but at a lexically enclosing level.
- The entire program heading, including the reserved word program,
  may be omitted.

**ERRONEOUS TESTS**

Number of tests attempted: 48
Number of errors detected: 11 (9 causes)
Number of tests failed: 2

Tests failed:
- Crash at run-time due to empty record-field.
- Relation between RTS I/O and implementation run-time support.

Errors detected:
- Assignment compatibility: index-type vs subscript value.
- Put not allowed if eof false.
- In(0.0) or In(negative)
- xrr(negative), but continues execution!
- trunc(largereal)
- round(largereal)
6.7.2.2-3 Div and mod by 0, but continues execution!!
6.9.2-4 Read of textfile, but cMrs do not represent integer value,
6.9.2-5 Read of textfile, but cMrs do not represent real value.

Errors not detected:
Use of undefined values.
Variant undefined.
All assignment compatibility except indextype in arrays.
Nil or undefined pointer dereferencing.
Undefined function result.
File 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.
Char 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-DEFINED TESTS

Maxint = 32767.
Set of char not implemented, but taken as equivalent to set of 'A'..'
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=0.24e-39; xmax=1.7e+08.

Full evaluation of boolean expressions.
Selection then evaluation in a[i] := exp.
Evaluation then dereferencing in p[i] := exp.
Writes two exponent digits in real numbers.
Default field widths integer 7
    boolean 5
    real 13

Rewrite permitted on output.
Both comment delimiters allowed, no others.

Run 16 implementation-defined tests successfully.

QUALITY TESTS
Number of tests attempted: 24
Number of tests failed: 3

Tests failed:
6.2.1-9 Compiler loops when presented with program with 50 labels.
6.6.6.2-7 Compiler refuses to compile a real expression in sin/cos test
due to "lack of registers".
6.6.3.9-6 Compiler crashes after compiling 11 nested for-loops.

Quality measurements:
5.7.9-1 6.1.9-6 Any length identifiers allowed; disallow all mis-spellings.
5.1.9-4 Unclosed comments swallow text without trace.
5.2.1-9 Allowable number of types ≥ 50.
6.4.3.2-4 Array[integer] diagnosed but message not good.
6.4.3.2-5 Record fields allocated representation space in declaration order.
6.4.3.2-6 Warshall's algorithm timing/space test not yet run.
6.5.1-8 Allowable number of variable declarations ≥ 100.
6.6.6.2-6, -7, -8 and -9 Quality tests on sqrt, arctan, exp and ln carried out.
Some minor inconsistencies.
6.9.1-11 Mod inconsistently implemented for negative operands.
6.6.3.9-5 No warnings for impossible case classes.
6.6.3.9-6 Allowable number of case-constants ≥ 256.
6.6.3.9-10 Undefined (out-of-range) values of case expressions are possible.
and are undetected but do no violent damage.
6.8.3.9-20 Allowable number of nested with-statements must be ≥ 10.
6.9.4-10 TextField without eol at end in still printed.
6.9.4-11 Recursive I/O allowed on same file, and still works.

-----------------------------------------------------------------
Pascal P4 on B6700 at Tas

PASCAL VALIDATION SUITE REPORT

Pascal Processor Identification:

Computer: Burroughs B6700
Processor: Pascal-P4 compiler received from Sydney, (March 1979) updated as described below.
The Pascal-P4 compiler was compiled with the B6700 Pascal version 2.9.001 (university of Tasmania) compiler.
The Pascal-P4 interpreter was compiled with the B6700 Pascal version 2.9.178.008 (UCSD) compiler.

Test Conditions:

Tester: C.D. Keen
Date: August 1979
Validation Suite: Version 2.2

Details of Update to Compiler

The original P4 compiler was updated to the level described in the Pascal Newsletter, 13 (Dec 1978).
The compiler and interpreter were both reformatted into 72 character lines.
The declaration of the file 'PRR' was included in the compiler.
The maximum length of constant strings, 'STRGLEN' was increased to 60 in both the compiler and interpreter.
The compiler was extended to accept both upper and lower case identifiers, with conversion to all upper case characters in 'INSYMBOL'.
The compiler was extended to recognise the alternative form of comment delimiters: '{...}'.
The sizes of the integer, real and set constant tables, and the bounds table in the interpreter were extended to 100 words each.

During the processing of the validation suite the following errors in the compiler were corrected:

a) In the procedure 'GEN2T' the field width of bounds variables was increased.
Replace: WRITELN(PRR,FPl:3+0RD(ABS(FPl) > 99)*5,FP2:8);
by: WRITELN(PRR,FPl:14,FP2:14);
b) The assignment 'FS := LSP' near the end of the procedure 'TYP' causes an error if 'LSP' is undefined. This can occur if the type commences with 'packed', but does not have the subsequent symbol in 'TYPEDELS'.
Replace: ERROR(10); SKIP(FSYS+TYPEDELS);
by: ERROR(10); SKIP(FSYS+TYPEDELS); LSP := nil
c) Fix number 17, proposed by Jim Miner (PN, Feb 1978, p71) to prevent comparisons of arrays and records within the procedure 'EXPRESSION' can result in 'TYPIND' not being defined. This also prevents comparisons of strings.

Replace: if not STRING(LATTR.TYPTR)
and (LOP in [LTOP,LEOP,GTOP,GEOP]) then
ERROR(131);
ERROR(134);
by: if not STRING(LATTR.TYPTR) then ERROR(134);
TYPIND := 'M';
Replace: ERROR(134);
by: ERROR(134); TYPIND := 'M';
d) If the bounds of a for statement are not parsed correctly then the name of the label at the commencement of the for statement will not be defined. This causes the generation of a jump to this label to crash at:
... GENULPXJP( ... ,LADDR); ...
To ensure that 'LADDR' is defined, even when syntax errors occur in 'FORSTATEMENT', insert the following statement at the commencement of the body of that procedure:
LADDR := INTLABEL + 1;

Conformance Tests:
Number of tests passed: 93
Number of tests failed: 46 (22 basic causes)

Details of failed tests:
Test 6.1.2-3 fails because only the first 8 characters of an identifier are significant, and not the identifier's actual length.
Test 6.1.7-2 fails because the maximum length of strings is restricted.
Test 6.2.2-3 fails because the domain of a pointer type T is not permitted to have its defining occurrence anywhere in the type-declaration-part in which it occurs.
Test 6.2.2-8 fails because assignment is not permitted to a function identifier at a lower level than the level at which the function is declared.
Tests 6.4.2.2-2, 6.7.2.2-5 and 6.8.3.9-7 fail because 'maxint' is not predeclared.
Tests 6.4.3.1-3, 6.4.5.1, 6.5.1-1 and 6.5.3.4-1 fail because declarations of files are not permitted.
Test 6.4.3.2-3 fails because the character set differed between the compiler and interpreter, so that integer checks on subranges of char are invalid.
Test 6.4.3.3-1 fails because variant parts without tag fields are not permitted.
Test 6.4.3.3-1 fails because a record containing only a semicolon produces a syntax error.
Tests 6.4.3.5-2, 6.6.3.1-3 and 6.9.4-15 could not be completed because 'text' is not predeclared.

Tests 6.4.3.5-3, 6.6.3.1-3, 6.6.5.2-3, 6.6.5.2-5, 6.9.1-1, 6.9.2-2, 6.9.2-3, 6.9.3-1, 6.9.4-1, 6.9.4-2, 6.9.4-3, 6.9.4-4, 6.9.4-6, 6.9.4-7 and 6.9.5-1 could not be completed because 'reset' and 'rewrite' are not implemented.

Test 6.4.3.5-4 fails because the output line is not flushed at the program's completion. This is not explicitly performed by the interpreter, but is dependent on the processor used to compile the interpreter.

Tests 6.6.3.1-5, 6.6.3.4-1, 6.6.3.4-2 and 6.6.5.1 could not be completed because procedure and function parameters are not permitted.

Test 6.6.5.3-2 could not be completed because 'dispose' is not permitted.

Test 6.6.6.3-1 could not be completed because 'round' is not permitted.

Test 6.8.2.1-1 could not be completed because the implementation of scalar types is dependent on the processor which compiled the interpreter, and the UCSD B6700 Pascal compiler does not correctly implement operations on boolean variables. (not is implemented as one's complement)

Test 6.8.2.4-1 fails because nonlocal goto's are not permitted.

Test 6.8.3.5-4 fails because the maximum range of case labels is 1000. (CIXMAX)

Test 6.8.3.9-1 fails because the assignment to a control variable in a for loop precedes the evaluation of the second expression in the for statement.

Test 6.9.4-7 fails because the writing of boolean variables to text files is not implemented.

Test 6.9.6-1 fails because 'page' is not permitted.

Deviance Tests

Number of deviations correctly detected: 52
Number of tests showing true extensions: 4
Number of tests not detecting erroneous deviations: 25 (13 basic causes)
Number of tests failed: 12 (6 basic causes)

Details of extensions:
Tests 6.8.3.9-9, 6.8.3.9-14 and 6.8.3.9-19 show that a for control variable can be globally declared.
Test 6.10-1 shows that file parameters in the program heading are ignored.

Details of deviations not detected:
Test 6.1.2-1 shows that 'nil' is a predeclared identifier, rather than a reserved word.
Tests 6.1.7-6 and 6.4.3.2-5 show that the index bounds of a string are not restricted to 1..n.
Tests 6.1.7-7 and 6.1.7-8 show that strings are permitted to be an array of a subrange of char.

Tests 6.2.2-4, 6.3-6 and 6.4.1-3 show that common scope errors are not detected.

Tests 6.3-5 and 6.7.2-2-9 show that a signed string is permitted as a factor.
Tests 6.4.5-2, 6.4.5-3, 6.4.5-4, 6.4.5-5 and 6.4.5-13 show that type compatibility is used with var parameters, rather than enforcing identical types.

Test 6.6.2-5 shows that no check is made to ensure that an assignment to a function identifier exists in the code of that function.

Test 6.6.4-6 shows that 'succ' and 'pred' can be applied to real arguments.
Tests 6.8.2.4-2 and 6.8.2.4-3 show that a goto is permitted between branches of an if or case statement.
Tests 6.8.3.9-2 and 6.8.3.9-4 show that an assignment to a for control variable is permitted within a loop.
Test 6.9.4-9 shows that an output field width can be negative.
Test 6.10-3 shows that 'output' can be redefined at the program level, and yet still exist as a file.
Test 6.10-4 shows that a program heading is not required.

Details of failed tests:
Tests 6.1.7-5 and 6.9.4-12 fail because the reserved word packed is ignored.
Tests 6.4.6-7 and 6.4.6-8 show that the determination of compatibility between set types is incorrect.

Error Handling

Number of errors correctly detected: 13
Number of errors not detected: 33 (16 basic causes)

Details of errors not detected:
Test 6.2.1-7 shows that variables are not preset to 'undefined' at interpretation time.
Tests 6.4.3.3-5 and 6.4.3.3-6 indicate that no runtime checks are performed on the tag field of variant records.
Tests 6.4.3.3-7 and 6.4.3.3-8 could not be completed because variant records without tag fields are not permitted.
Tests 6.4.6-7 and 6.4.6-8 show that the determination of compatibility between set types is incorrect.
Test 6.6.2-6 shows that the use of a function without assignment to the function-value-variable is permitted.
Tests 6.6.5.2-1, 6.6.5.2-2, 6.6.5.2-6, 6.9.2-4 and 6.9.2-5 could not be completed because 'reset' and 'rewrite' are not implemented.
Test 6.6.5.2-7 could not be completed because declarations of files are not permitted.
Tests 6.6.5.3-3, 6.6.5.3-4, 6.6.5.3-5 and 6.6.5.3-6 could not be completed because 'dispose' is not permitted.
Tests 6.6.6.2-4, 6.6.6.2-5 and 6.7.2.2-3 show that no explicit checks are made by the interpreter for invalid arguments to 'alog', 'sqrt', '/', div and mod.
Test 6.6.6.3-2 shows that no check is made to ensure that the result of a call to 'trunc' is in the range -maxint..maxint.
Test 6.6.6.3-3 could not be completed because 'round' is not permitted.
Tests 6.7.2.2-6 and 6.7.2.2-7 could not be completed because 'maxint' is not predeclared.
Test 6.7.2.4-1 fails because operations on overlapping sets are not detected.
Tests 6.8.3.9-5 and 6.8.3.9-6 show that the use of a for control variable immediately after the loop termination is not prevented.
Test 6.8.3.9-17 shows that two nested for statements with the same control variable are permitted.

Implementation defined
Number of tests run: 15
Number of tests incorrectly handled: 9 (5 basic causes)

Details of implementation-dependence:
Only implementation details fixed in the compiler are considered, and not details dependent on the processor which compiled the interpreter.
Test 6.4.2.2-7 shows that 'maxint' is not predeclared.
Tests 6.4.3.4-2 and 6.4.3.4-4 show that the maximum permitted range of set values is 0..47. This is the minimum required to parse the P4 compiler.
Test 6.6.6.1-1 could not be completed because procedure and function parameters are not permitted.
Tests 6.7.2.3-2 and 6.7.2.3-3 show that boolean expressions are fully evaluated.
Tests 6.8.2.2-1 and 6.8.2.2-2 show that a variable is selected before the expression is evaluated in an assignment statement.
Tests 6.9.4-5, 6.9.4-11 and 6.10-2 could not be completed because 'reset' and 'rewrite' are not implemented. The default field width specifications are: 10 for integers, 20 for reals (floating point notation only) and 1 for characters.
Tests 6.11-1, 6.11-2 and 6.11-3 show that no alternative symbols have been implemented.

Quality Measurement
Number of tests run: 23
Number of tests incorrectly handled: 11

Results of tests:
Test 5.2.2-1 shows that only the first 8 characters of an identifier are significant.
Test 6.1.8-4 shows that no warning is given of semicolons within comments.
Tests 6.2.1-8, 6.2.1-9 and 6.5.1-2 indicate that long lists of declarations can be made in each block, depending on the maximum extent of the heap in the compiler.
Test 6.4.3.2-4 shows that an array with 'integer' index type is not permitted.
Test 6.4.3.3-9 shows that variant fields of a record occupy the same space, using 'reverse correlation'.
Test 6.4.3.4-5 (Warshall's algorithm) took 172.530 seconds CPU interpretation time. (cf. 10.498 seconds CPU execution with UCSD B6700 compiler)
Test 6.6.1-7 shows that the maximum depth of procedure nesting is 10. (MAXLEVEL)
Tests 6.6.6.2-6, 6.6.6.2-7, 6.6.6.2-8, 6.6.6.2-9, 6.6.6.2-10 and 6.7.2.2-4 are only relevant to the implementation of 'sqrt','atan','exp','sin','cos', 'ln', div and mod in the processor which compiled the interpreter.
Test 6.8.3.5-2 shows that case constants must be compatible with the case index, but do not have to be of the same type if the case index is a subrange.
Test 6.8.3.5-8 shows that a large case statement is permitted.
Test 6.8.3.9-18 shows that a range check is performed in a case statement after a for statement to check the value of the for control variable.
Test 6.8.3.9-20 and 6.8.3.10-7 show that for and with statements can be nested to a depth exceeding 15.
Test 6.9.4-10 shows that the flushing of the output line buffer depends on the processor which compiled the interpreter.
Test 6.9.4-14 shows that recursive I/O is permitted, using the same file.

Extensions
Number of tests run: 1
Details of test:
Test 6.8.3.5-14 shows that no extensions have been made to the standard syntax of the case statement.
Stamp out Bugs

Pascal Validator

(* Hurry and send a Validation Report on the Pascal Compiler you use to Pascal News! *)
POLICY: PASCAL USER'S GROUP (79/09/01)

Purposes: Pascal User's Group (PUG) tries to promote the use of the programming language Pascal as well as the ideas behind Pascal through the vehicle of Pascal News. PUG is intentionally designed to be non-political, and as such, it is not an "entity" which can take stands on issues or support causes or other efforts however well-intentioned. Informality is our guiding principle; there are no officers or meetings of PUG. The increasing availability of Pascal makes it a viable alternative for software production and justifies its further use. We all strive to make using Pascal a respectable activity.

Membership: Anyone can join PUG: particularly the Pascal user, teacher, maintainer, implementor, distributor, or just plain fan. Memberships from libraries are also encouraged.

See the ALL-PURPOSE COUPON for details.

FACTS ABOUT Pascal, THE PROGRAMMING LANGUAGE:

Pascal is a small, practical, and general purpose (but not all-purpose) programming language possessing algorithmic and data structures to aid systematic programming. Pascal was intended to be easy to learn and read by humans, and efficient to translate by computers. Pascal has met these design goals and is being used quite widely and successfully for:

* teaching programming concepts
* developing reliable "production" software
* implementing software efficiently on today's machines
* writing portable software

Pascal is a leading language in computer science today and is being used increasingly in the world's computing industry to save energy and resources and increase productivity.

Pascal implementations exist for more than 62 different computer systems, and the number increases every month. The Implementation Notes section of Pascal News describes how to obtain them.

The standard reference and tutorial manual for Pascal is:

Pascal - User Manual and Report (Second, study edition)
by Kathleen Jensen and Niklaus Wirth
1978 (corrected printing), 167 pages, paperback, $7.90.

Introductory textbooks about Pascal are described in the Here and There Books section of Pascal News.

The programming language Pascal was named after the mathematician and religious fanatic Blaise Pascal (1623-1662). Pascal is not an acronym.

Pascal User's Group is each individual member's group. We currently have more than 3357 active members in more than 41 countries. This year Pascal News is averaging more than 120 pages per issue.