Return to:
Pascal News
2903 Huntington Rd. • Cleveland, Ohio 44120
Return postage guaranteed Address Correction requested

Attn: Pascal Group [83]
Univ. of Minnesota
Room 217 Browse Copy
UCC: 227 EX
Minneapolis,MN 55455

Rm 217 Browse Copy $10.00

PASCAL USERS GROUP

Pascal News

Communications about the Programming Language Pascal by Pascalers

- Pascal Processor Validation Procedure
- A Better Referencer
- Use of Generic Capsules
- Implementation Reports
- Validation Suite Reports
- Announcements

Number
25
APRIL 83

POLICY: PASCAL NEWS (Jan. 83)

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

Purpose: The Pascal User's Group (PUG) promotes 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 takes 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 COUPON for details.

- *Pascal News* is produced 4 times during a year; January, April, July October.
- ALL THE NEWS THAT'S FIT, WE PRINT. Please send material (brevity is a virtue) for *Pascal News* single-spaced and camera-ready (use dark ribbon and 15.5 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 — explains 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.

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.

VALIDATION SUITE REPORTS — reports performance of various compilers against standard Pascal ISO 7185.

Pascal News

Communications about the Programming Language Pascal by Pascalers

APRIL 1983 **Number 25**

Good Members Hello;

I now have control of most elements of Pascal News and future submission articles, comments and good jokes should be addressed:

Pascal News
2903 Huntington Road
Cleveland, Ohio 44120

Our United Kingdom and European elements are thriving and boisterous.

PUG (UK)	PUG (Eur)
P.O. Box 52	ARGE Pascal
Pinner	Hellmut Weber
Middlesex HA5 3FE	Degenfeldstrasse 2
U.K.	D — 8000 München 40

These groups should be excellent sources of local and international information.

We have lost an element and have no successor. PUG (Aus) has experienced increased costs and decided PUG (USA) could support them with little loss of timeliness. I would like to thank them for their past performance. I am sorry I did not have the opportunity to work with them.

PUG (USA) will now serve various needs.

We now serve inside USA and outside USA members and also provide an air mail option for those who need Pascal News as quick as possible.

Writing of timeliness I am reminded that the newsletter has deadlines. These are January 1st, April 1st, July 1st and October 1st. When you have material for the newsletter please send it as quick as possible. Do not worry about the deadlines but keep in mind news loses its value as it matures.

I will continue to publish implementation notes and announcements of the trade. I encourage members and vendors to test drive their new compilers with the "Validation Suite". Send the reports to me and then we will all know the best performing compilers.

I have been asked if we would pay for articles. I have thought about this and worried where I would get the money.

I have decided to accept advertising and use this money to pay honorariums to writers of good articles.

A reminder that back issues will reflect higher reprinting costs and have a $25 per set price after July 1st

Its still a bargain at $15 now.

One more thing. Thank you for your renewals and lovely comments. I have been encouraged by your thoughts.

Charlie

Dear Pascalers,

here we are reopening PUG Europe:
Lor + Martha +
Erwin + Hellmut +
Jurgen + Manfred +
Urf (Korbinian).

We are Pascal fans and users from the university and industry who are organizing in our spare time the distribution of Pascal News for the European region.

From our viewpoint, being mainly Pascal users, we would like to encourage you to help in keeping Pascal News a living forum, a market place for all Pascal users. So here again is a call for papers and programs. There are certainly many tools, especially for textprocessing, which are of interest for the Pascal community, maybe for simple use, maybe in order to compare ideas about problems which many of us may already have encountered. And think about all the programs for solving the daily commercial problems.

Another subject which we think important is documentation. If you have to (or like to) use non-standard features mentioning those increases portability. An extreme example of the necessity of documentation are two Pascal implementations which use one name (append) for two non-standard predefined procedures doing different things (append one string to another versus open a textfile for appending text).

To increase the market place function of Pascal News we should like to ask everybody who provides a Pascal source for publication to state whether he/she is willing and/or able to distribute this source in machine readable form (or even as a well readable listing) and if so at what cost.

Lastly we would like to ask all those wishing to contact us to use our official address:

ARGE Pascal
Hellmut Weber
Degenfeldstrasse 2
D — 8000 München 40

and *not* to send registered letters. (We had some problems, as there is no Mr. Pascal to claim them.) If you want to send us money for subscription please use our postgiro account Munchen 51589-801 or send an Eurocheque and please take note that any other form of payment means additional paperwork for us.

Stay happy with Pascal!

Pascal Users Group (U.K.)

Pascal News 23a is a supplement, to plug the lengthening gap between US originating 23 and 24. Readers will note that its contents are quite different from those of previous editions. There is a shift of emphasis from matters of concern at leading edge University level, to those of concern to producers and users of inexpensive standardized products.

That shift has been wholly dictated by the content of material submitted for publication. Whether it is a temporary side-step or a permanent change, will also be decided by contributors (to future editions). PUG(UK) is the servant of you the subscribers and as such, will publish material originating from any section of the user community.

We are all indebted to each contributor but Tony Heyes's generosity in offering his Bibliography suite of programs for refinement through the medium of PN is particularly appreciated. Constructive critiques are welcome.

There is a widening of the user base and an overdue deployment of resources to that end, evidenced by the complementary nature of articles from widely differing sources. Read on and judge for yourselves. Although you will find that 23a is pitched at quite a different level from that of your usual expectations of PN, I sincerely hope that you will welcome it as a stop-gap until 24 becomes available from Rick, Andy, and Co.

The following is offered as an illustration of the scene which prompted the production of a supplement.

Intrigued by advertising which referred to "mere humans", I went along to the personal computer show at the Barbican on September 12th.

Imagine the disappointment at failing to find anything innovative or even mildly interesting. Discovered that with a single exception, exhibitors did not know whether standard Pascal was implemented on the machines offered to the public. More than one of those asked, replied "Yes, it's called UDCS or something like that". At one stand, sponsored by British Petroleum, the Department of Trade and Industry, the Council for Educational Technology, and others, an 'expert' merely looked blank and suggested that I ask someone else. 'Someone Else' replied "We are only interested in things for use in Education". At the National Computing Centre stand, another expert, when asked if his stand offered any information about standard Pascal and its implementation or use in a microcomputer environment, replied "No, there is no demand", deftly followed by "Can I help you sir?" to someone standing behind me. In some instances, the initial answer was "Yes", followed by misrepresentative flannel when a demonstration was requested.

Met a guy who holds a powerful position in the largest education authority in Britain. He believes that BASIC is an "appropriate" language for the "mass" of young people who "won't bother" to become seriously interested in the technology. I should admit at this point, that had my first experience of a perception of machine intelligence been through the medium of BASIC (or COBOL, FORTRAN, etc.), I might easily have joined the ranks of those who either "won't bother" or are suitably unimpressed by obscure combinations of hunches, guesses, and a dash of perceptual skill which only occasionally fails.

PUG

I.T. and M.I.S.S.

From Wireless world.
Reproduced with Phillip Darrington's permission

One of the aims of Information Technology Year and the Microelectronics Education Programme is to involve schoolchildren in the use of microcomputers and related electronic devices. There are the M.E.P., the Micros in Schools Scheme, exhibitions and events throughout the year and beyond. It is, perhaps, fortunate that Mr. Callaghan happened to be watching television on the evening the programme "Now the Chips are Down" was broadcast and was spurred into action then, or we would probably find the propaganda even more frenetic than that now being put out by the energetic Mr. Baker, the prophet of IT.

Information Technology is a curiously diffuse name for a Year. The official definition, "the acquisition, processing, storage, dissemination and use of vocal, pictorial, textual and numerical information by a microelectronics-based combination of computing and telecommunications" appears to encompass most of the activities of the average person, except eating and one or two other processes, although the use of a computer is not often considered essential to the more basic of these.

So far as its involvement of schoolchildren is concerned, the publicity is decidedly shrill, the Minister's aim being to have a computer in every secondary school by the end of the year and even to think about providing them for primary schools.

There can be no argument that young people must be aware of computers and how to use them, but it does seem possible that the present blaze of publicity tends

to obscure the point that computers are a means, not an end. There is also the question of how the micros are to be used in schools.

According to the fifth edition of the Concise Oxford Dictionary (now, admittedly, modified), a computer is "a calculator — an electronic calculating machine" — an unfortunate description, taken too literally by at least some of those responsible for introducing youngsters to computing, with the result that the school micro is often given to the senior math teacher to guard with his life, presumably on the grounds that computers are electronically mathematical and possess no relevance to any other subject.

In other schools, the computer is treated as a kind of totem, and the pupils are taught "Computer Studies". As a subject, computing (meaning programming) is a singularly empty one, unless the pupil learning it intends to become a programmer. A computer is an aid to the process in which it is used — in this instance, learning — and an element of transparency to the user rather than an obscuring of the subject by undue attention to the computer must be the aim.

Clearly, an overnight transformation, after which every teacher would be using a micro as to the manner born, is hardly feasible. But, until the school micro (or one of its terminals or even a micro owned by a pupil or teacher) can be used naturally, as is a dictionary or pocket calculator or a video recorder, it will dominate the learning process. Utmost priority should be given to teachers from all disciplines, from home economics to athletics, to use the computer as an aid, rather than as a distraction, so that pupils who are not to specialize in science or engineering can see that it is of advantage to them to be at ease with computers, but no more than that.

The Inner London Education Authority is aware of these problems and is educating teachers in the use of computers so that, even though there may be only one micro or terminal in the classroom, the pupils will learn the place of a computer by, to use ILEA's word, "osmosis". However, there is evidence aplenty that education authorities in other areas are either hypnotized or revolted by the new equipment and, accordingly, either enshrine it or pass it to the school computer fanatic to impress people with.

In short, a computer is a useful tool, but that is all it is: it can help or it can dangerously hinder learning, and only the education of teachers in its natural use as an aid can decide which. **PUG**

Pascal — An Effective Language Standard

Brian A. Wichmann,6/5/82

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

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

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

1. The Pascal standard is more comprehensive than that of COBOL or FORTRAN. For instance, the COBOL and FORTRAN standards do not require that an invalid program is rejected by a compiler. The Standard for these languages is just a definition of a language rather than a set of requirements for a compiler. This is clearly not very satisfactory since we all write incorrect programs on occasions.
2. The Pascal Standard is simple and devoid of a multitude of options. If the language has lots of options, then program portability is reduced because a program may not be valid without a specific option. COBOL has a large number of options and FORTRAN 77 has two major levels (essentially distinct languages) whereas Standard Pascal has just one option, affecting only one part of the language. This option is to allow procedures to handle arrays whose size varies from call to call. This option, level 1 Pascal, would allow Pascal programs to call FORTRAN routines in many systems.
3. The Pascal test suite is more searching than that of COBOL and FORTRAN. This is essentially a consequence of the definition of the language. The National Physical Laboratory has been collaborating with the University of Tasmania on the construction of this suite for over two years. About 400 copies of the test suite have been sold worldwide. A new version of this suite has recently been issued to correspond to the new ISO Standard. Unlike the COBOL and FORTRAN test suites, the one for Pascal includes incorrect programs which must be rejected: ones to examine the error-handling capability of a compiler, and the "quality" of an implementation. The quality tests indicate if there is any small limit to the complexity of programs that a system can handle and also assesses the accuracy of real arithmetic.

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

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

The last step in this process is to encourage users to request a *Standard* compiler from the suppliers and for suppliers to meet that demand. As a contribution to this last step, NPL held a conference on this topic with its collaborators. Professor Arthur Sale from the University of Tasmania addressed the conference making it an international event. The other key speakers were John Charter from BSI who described how a validation service run by BSI would work. Professor Jim Welsh from UMIST who described how the Standard can be implemented and Lyndon Morgan from NCC who described a guide written to support the test procedures. Also Barry Byrne, from ICL explained how the provision of a standard compiler for Pascal is advantageous in both marketing and for internal use. Mr. Ken Thompson from the European Commission explained the usefulness of international standards within the Community and some of the problems in their effective exploitation.

This program contains five errors, often undetected by compilers. Can you spot them?

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

Try it on your system and see how many errors are detected.

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

The corrected program is:

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

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

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

All the marks are out of 5. The half marked for detecting an error indicates that the error message was confusing enough for it to be unclear if the error was properly detected. Naturally, the last two columns are subjective. **PUG**

Article formed the basis of piece in *Computer Weekly* by Phillip Hunter. 11th Feb. 1982 page 14

PASCAL PROCESSOR VALIDATION PROCEDURE

By David Blyth
Standardization Office,
National Computing Centre

1 Introduction

Few Pascal users can be unaware of the recent publication of the British Standard for the language which will shortly be adopted internationally. Many users have heard of the suite of validation programs, developed by the University of Tasmania and the National Physical Laboratory, which can be used to check on the standard-conformance of an implementation. This suite is readily available and any user who has a copy can use it to test his own compiler or interpreter. For those brave users who undertake such testing this article presents a brief guide to the steps involved and draws upon experience gained at NCC in a joint NPL/NCC/BSI project to develop and document the validation procedures.

2 The Pascal Standard and Validation Suite

The Pascal standard defines the language itself and the manner in which Pascal programs are to be handled by an implementation. The validation suite contains over 400 test programs whose purpose is to check whether or not an implementation accepts the language as defined in the standard and whether or not programs which are accepted behave as the standard says they should. The standard and the validation suite have been developed in parallel with the result that the suite will provide an exceptionally strenuous test of any implementation. An implementation which performs well under test can be used with confidence in its conformance and reliability.

The suite contains eight types of test program which investigate respectively, conformance, deviance, implementation-defined features, implementation-dependent features, error handling conformance arrays, quality and extensions. These classes of tests are quite distinct and are used in characteristic ways.

2.1 Conformance Tests

Conformance test programs attempt to check that an implementation provides those features required by the standard and that it does so in the manner which the standard specifies. These programs are all correct standard Pascal. If the implementation conforms to the standard these programs all compile and execute. If a conformance test program fails then it is an indication that the implementation does not conform to the standard.

2.2 Deviance Tests

Deviance test programs check whether

(i) the implementation provides an extension of Pascal;
(ii) the implementation fails to check or limit in an appropriate manner some feature of Pascal;
(iii) the implementation incorporates some common error.

No deviance test program is standard Pascal. Each such program contains exactly one such deviation. When a deviance test is run the results are inspected for evidence that the implementation does in fact detect the deviation. If it does not then the implementation does not conform with the standard.

2.3 Implementation-Defined Features

The standard defines an implementation-defined feature as one which may differ between implementations but which is defined for any particular processor. A conforming implementation must be accompanied by a document that provides a definition of all its implementation-defined features. The test programs for implementation-defined features are intended to show how these features are handled in any particular implementation. If they aren't handled in the manner claimed then the implementation does not conform.

2.4 Implementation-Dependent Features

An implementation-dependent feature may differ between implementations and is not necessarily defined for any particular implementation. Here the implementor can either state in his documentation that use of such features is not reported or else have the implementation issue some diagnostic for which such a use is encountered. The test programs in this area are designed to determine the behaviour of the implementation. The implementation conforms only if it behaves as claimed or reports implementation-dependent usages.

2.5 Error-Handling

An error is defined, in section 3.1 of the standard, to be a violation by a program of the requirements of the standard that the implementation is not obliged to detect. An implementation only fails to conform in respect of error-handling if it fails to process an error in the manner claimed in the documentation. The error-handling tests each present the implementation with one error with the aim of determining exactly what the implementation does with it.

2.6 Conformant Arrays

An implementation may conform with the standard at level-0 or at level-1. In plain terms it can either have conformant arrays or it can't. If conformant arrays are provided then all of the features specified for them must be provided according to the standard.

The conformant array tests are a collection of conformance, deviance, implementation-defined, implementation-dependent, error-handling and quality tests designed to test the conformant array features in isolation.

2.7 Quality

Many aspects of an implementation are beyond the scope of the standard, but it is still useful to investigate them. Quality tests explore these areas and investigate:

(a) The limits on the size and complexity of programs imposed by the implementation
(ii) the amount of store needed to perform certain well-defined tasks
(iii) the accuracy of real arithmetic
(iv) the meaningfulness of diagnostics for common types of error
(v) the speed of the code produced.

Quality tests often throw up some surprising results!

2.8 Extensions

Many implementations offer extensions to the standard. The extension tests see whether common extensions (eg those approved by PUG) are implemented.

Together the test programs provide a very thorough test of an implementation.

3 Using the Validation Suite

3.1 Distribution Format

The validation suite is distributed on 9 track magnetic tape with characteristics as follows:

Recording density : 800 or 1600 bpi
Recording mode : NRZI or PE
Character code : ISO 646 or EBCDIC
1200 bytes/block, 80 characters/record.

A purchaser of the tape can specify which density, recording mode and character code he wants.

There are 49 files on the tape. Three of these contain documentation. The rest contain the validation programs.

3.2 Media Conversion

Users whose machines have tape drives should experience no significant problems in reading the distribution tape. Their only concern will be with lexical conversion if necessary.

Users with floppy disc based systems need to do a media transcription to get the suite in a form in which they can use it. This conversion can be tricky, and is almost always done on an ad hoc basis for the particular system concerned.

3.3 Lexical Conversion

There are two character sets to consider when using the suite — the one used to encode the test programs, and the one used to represent "char-type" values on the target computer.

Roughly speaking any consistent set of lexical substitutions can be made, but some may render specific lexical test programs, and some programs which test the char type, irrelevant in validation.

Care is needed to ensure that lexical conversion is consistent throughout. This is particularly important if media conversion affects character code representations.

3.4 Integrity Checking

Following media and lexical conversion it is advisable to check that no corruption has occurred. For this purpose a program called the Checktext program is supplied. It produces a 96-bit binary check pattern using an algorithm originally developed for use in data transmission (CCITT Rec. V.41)

The Checktext program operates on a standardized internal representation of the program and will not be affected by legal lexical substitutions. Certain parts of the program may need customization for use on particular systems and the source code is marked to show where such changes should be made.

The results of the Checktext program should be compared with standard results contained in the User Guide to the suite (supplied with the distributrion tape) and if there is any discrepancy then transcription has introduced errors.

3.5 Checking Validation Suite Assumptions

A validation suite must necessarily make certain assumptions about the nature of the implementations which it will be used to test. The Pascal validation suite assumes that

- text files
- character-strings
- the real-type
- local files

are all implemented, also that

- lines up to 72 characters long can be accepted
- lines up to 72 characters long may be output
- the value of maxint is > 32,000
- the relative precision for reals is < 0.001
- the characters needed to encode the test programs are all accepted as distinct by the implementation
- the "largest" procedure in the test suite is accepted by the implementation (except for certain quality test procedures).

A further implicit assumption is that the real arithmetic system is susceptible to investigation by certain types of method.

The validation suite contains a program called the "Check Assumptions" program which enables the user to determine whether or not the implementation violated any of the assumptions listed above.

4 Planning and Running the Tests

4.1 Planning is Important

Testing an implementation is not just a matter of running all the test programs. The test suite is large and on some machines it is not possible to run all the tests without breaking the suite into batches. Furthermore close attention must be paid to ensure that the behaviour of the implementation is accurately recorded throughout the test procedure. Finally provision must

be made to make it easy to re-run any particular test after preliminary interpretation of test results.

Choice of the method of working can have a marked effect on the overall time taken to run the tests. There are two areas to consider. First some method must be chosen to extract test programs from the files which contain them. Second the organization of the jobs which run the test programs must be decided. The User Guide illustrates three approaches for each of these methods which will cover most cases on a wide range of machines.

Some programs may prove to be rogues on certain implementations. There is no way of knowing in advance which programs will behave in this way for any given implementation. The user should take care so that such programs do not cause the loss of accumulated test results.

In any event some programs will need re-running because the results on the first run may have been inconclusive. The circumstances in which a re-run is needed are given in the Guide.

5 Reporting Results

It is desirable to adhere to a standard form of presentation when reporting the results of a validation. This offers two main advantages.

First, when a formal validation is being done, a standardized report:

1 Processor Identification
2 Test Conditions
3 Conformance Test Results
4 Deviance Test Results
5 Error-Handling Test Results
6 Implementation Defined Test Results
7 Implementation-Dependent Test Results
8 Level 1 Test Results
9 Quality Test Results
10 Extension Test Results

Guidance on the content and presentation of these sections is included and a sample validation report is included as an Appendix.

6 Practical Use

The present article offers only a brief sketch of the validation procedure. At first sight it may look somewhat daunting. In practice the key is attention to detail. The User Guide gives fairly detailed advice on transcription and test job organization, and will be found helpful by most people undertaking tests of implementations. Once transcription and organization have been sorted out the tests usually run smoothly. Carrying out a full test is a rewarding exercise which offers many lessons to language implementors. It is hoped that users and implementors alike will use the test suite and help to promote rapid practical standardization of Pascal.

PUG

Dear Nick,

After our phone conversation the other week, I was rather more relieved to feel that here in the UK there are other Pascalers at work and that PUGUK is viable again. The gap has been too long, and I wish you well in trying to get it going again. I shall try and do what I can and particularly with public domain software, but at the moment, I don't have a great deal of time to spare, nor any telecomms equipment to plug into my computer.

I enclose a cheque for 9 pounds for subscription. On the question of back numbers, I have copies of 12-16, and any subsequent or previous issues would be very welcome. I would have thought that for 17-21 which you already have, it would be worth while putting a note in the next issue to see how many people want them, and then have your printer print adequate copies in total. Much better than spending your time collating everyones' needs and doing individual photocopies of bits and pieces. Perhaps if other people were able to lend you some of the older copies, the same could be done. I'd certainly lend you 12-16 if you like. After all, its the information that matters, not whether the issue is an original or not unless we have an collectors among us. Anyway, mark me down for any back issues you can get your hands on, please.

I am now using Pro-Pascal from Prospero Software as my major programming tool, as well of course as Wordstar to compose programs and write letters. The hardware is OEM kit from Sirton Computers in Purley, by the name of Midas and is in essence an Integrand 10-slot S100 case with PSU, Ithaca IEEE S100 cards (MPU-80, FDC-2, 64KDR and VIO boards) giving 64k and 4Mhz Z80A with CP/M, plus 2*YE-DATA 174D 1Mb drives. The printer is a Qume (a luxury really), and a Volker-Craig VC404 completes the outfit.

I will try and compose a critique of Pro-Pascal as soon as possible, but version 1.4 is due out soon with 8 byte longreals among other goodies. I have written to Charles Foster of Pascal/Z User Group asking if he or his contributors would permit the distribution of any of their Pascal sources to PUGUK members appropriately modified to BS 6192, or if indeed there is any other Public Pascal around in the States. I think we ought to be prepared to reciprocate on this, don't you?

In converting from programming mainly on maintrames in Fortran and having a nodding acquaintance with Cobol, Basic and other languages, there are times when even Standard Pascal has its limitations. Therefore, I've thought of two ways of improving the language. As PUG may have some influence with the powers that be, I've taken the liberty of including the suggestions — by all means put them in a news-letter if you like. I don't believe in trying to persuade compiler-writers to augment their compilers as their job is to implement the standard. If the language is to grow, and if any such need is identified, then it's the standard that must mature. Now BS 6192 is published, it will be some time before any further thought is applied to the subject I expect, if ever, so perhaps now is the time to see if anyone is interested.

John R Logsdon
18 Darley Road
Manchester M16 ODQ

Tongue-in-cheek Pascal Language enhancements.

a) Structured constants.

Program make-up to be for example:

```
PROGRAM example;
CONST onehundred=100;
      ..................... etc
TYPE
  scalartype=(coffee,jam,bread,tea,biscuit,suicide);

      extype=RECORD
               a:integer;
               b,c:char;
               d:array[0..3] of integer;
               f:scalartype;
               g:set of scalartype;
               h:array[1,.20] of char
             END;
                    ..................... etc
TABLE ex1:extype=
            onehundred,'a',chr(20),(0,25,50,75),jam,
            [coffee,tea,bread],'cholesterol';

VAR exvar:extype;displayl:char;

BEGIN
  exvar:=ex1;
  displayl:=ex1.h[4];

      ..................... etc
```

Note the use of the 'chr' function to set up unprintable characters, the absence of any delimiter other than those already used in Pascal and the access of a constant array element. There is no reason why 'ord' should not also be included so that portability is enhanced. The syntax follows closely on that of Pascal as it is and involves no ambiguity in type declaration implicit where structured constants are declared in the constant section as in some implementations. Pointers declared in the correspnding type declaration may be set to whatever internal value represents nil, however they are named and uncompleted arrays of char initialized to spaces.

Such a feature will provide geniune structured read-only constants without the ugly initiation presently necessary in Pascal. In fact, in practice it is easier to put records for initialization in a parameter file and read them in, which does not seem an elegant solution. For micros with restricted memory, initializing a record from constants needs up to two copies of every element — one dynamic and one in the constant area, which is rather wasteful of space.

b) Type-change function.

Syntax to be, for example:

```
PROGRAM another;

CONST ..................... etc

TYPE score=(first,second,third,fourth);
     fruit=(apples,pears,oranges,grapes);

VAR thisscore:score;thisfruit:fruit;

BEGIN

    {calculate thisscore somehow}

  thisfruit:=fruit(thisscore);

      ..................... etc
```

This facility will provide a logical completion to the built-in functions 'ord', 'chr' and provide a much more readable alternative to the use of variant records. Although there is no reason why the method should not be available for records if the matching of record lengths were entirely the programmers responsibility, there is an objection in that the internal representation of variables will be machine-dependent. I envisage this type-change function purely for scalar variables between scalars and perhaps for pointers between pointers. It is of course really a mechanism to cause the compiler not to check types.

(This facility is similar to one available in AAEC Pascal 8000 for the IBM 360/370 series, and attributed to Kludgeamus)

If any readers have any comments for or against, perhaps PUG can help to air views?

HELP!

Dear Nick;

Systems Used

(i) Apple (II) UCSD Pascal.
(ii) To be delivered December 1982: Burroughs B21-5 (384 K Byte). Pascal ISO draft 5.

Special Interests

Business systems. Particularly rapid access to unsorted data items. Data base management systems.

Information Please

We would be interested in knowing of a Pascal compiler to interim ISO standard or UCSD for Burroughs B1955 with 0.5M Byte working store. Manufacturer does not support Pascal for.

Mr. P A E Herring
MAPAC
17 Market Square
Leighton Buzzard
Bedfordshire
LU7 7EU

Dear Nick,

CET TELESOFTWARE PROJECT

Thank you for your letter of 6th December.

I think you must have got the wrong impression from my letter of 3rd December. We certainly do not want to see a different telesoftware format for PASCAL. As I understand it, the only problem with the cur-

rent format is the TAB character which lies outside the PRESTEL character set. You may be interested in our recent extensions to the format (copy enclosed) which overcome this.

As far as including PASCAL programs in our library is concerned, all I am saying is that we need to learn how to walk before we can run. We are keen to include programs in languages other than BASIC, including PASCAL, but need to be sure there are people who can receive them on our system and will find them useful, before putting them up.

If you know of PASCAL programs which will run on the micros most used in educations, ie 380Z, Apple, Pet, Acorn and TRS 80, I would be interested in receiving details.

Chris Knowles
Telesoftware Project Manager
Council for Educational Technology
3 Devonshire Street, London W1N 2BA

Dear Pascal User,

Please find enclosed details regarding Version 3.1 of the Pascal Validation Suite which was released on the first of October 1982. Should you wish to receive a copy of the suite, please fill in the enclosed application form for a license and send it together with your remittance to:

Dr. Z. J. Ciechanowicz
Division of Information Technology & Computing
National Physical Laboratory
Teddington
Middlesex TW 11 OLW England

On receipt of the form and remittance we will send a magnetic tape containing the suite.

The cost of the package is $100 sterling (+ 15% VAT for UK users) and cheques should be made payable to "The National Physical Laboratory" quoting our reference number NPS 2/01.

Z. J. Ciechanowicz
Division of Information Technology & Computing
Department of Industry
National Physical Laboratory
Teddington, Middlesex TW11 OLW

PS When requesting the suite please supply the tape format you require:
i.e. 1600/800 b.p.i.
ISO/EBCVDIC code

We generally write our tapes with fixed length blocks, 15 records per block, 80 characters per record.

Dear Nick,

1. Can you recommend a PASCAL for XENIX? (LSI II UNIX)
2. Do you know who distributes the Dutch 'Fres University' version of PASCAL? (in the UK)

Brian Kirk
Robinson Systems
Engineering Limited
Red Lion House, St. Mary's Street,
Painswick, GL6 6QR
Telephone: (0452) 813699
VAT Registration: 302 3124 28

APPLICATION FOR LICENSE TO USE VALIDATION SUITE FOR PASCAL

Name and address of requester (company name if requester is a company)

Name and address to which information should be sent (write 'as above' if the same)

Signature of requester

Date

In making this application, which should be signed by a responsible person in the case of a company, the requester agrees that:

(a) The copyright subsisting in the validation suite is recognized as being the property of the British Standards Institution and A.H.J. Sale;

(b) The requester will not distribute machine-readable copies of the validation suite, modified or unmodified, to any third party without permission, nor make copies available to third parties.

In return, the copyright holders grant full permission to use the programs and documentation contained in the validation suite for the purpose of compiler validation, acceptance tests, benchmarking, preparation of comparative reports, and similar purposes, and the provision of listings of the results of compilation and execution of the programs to third parties in the course of the above ctivities. In such documents, reference shall be made to the original copyright notice and the source.

OFFICE USE ONLY

Signed

On behalf of A.H.J. Sale and the British Standards Institution

---

National Physical Laboratory Teddington Middlesex TW11 OLW Telephone 01-977-3222 Telex 262344

Pascal Compiler Validation Suite

NPL issued version 3.1 of the above suite of test programs on 1 October 1982. These programs permit a user to check the compliance of a Pascal compiler and run-time system with the ISO standard for Pascal (ISO 7185, also BS 6192). The new suite is an extensive revision of version 3.0 and the work has been undertaken in conjunction with Professor A.H.J. Sale of the University of Tasmania. Subsequent revisions to the test suite are likely to be of a minor nature.

The British Standards Institution will shortly be launching a pilot validation service base upon the test suite together with other material.

The test suite consists of about 17,300 lines of Pascal programs plus addition comments on each of the 553 test programs. The programs themselves are divided into a number of classes as follows:

182 programs checking that the features of the Standard are available;
157 programs checking that illegal constructs are rejected by a compiler;
82 programs checking the error-detection capability of a Pascal system;
60 programs checking the quality of an implementation;
40 programs checking for Level 1 Pascal ('conformant arrays')'
16 programs checking the variations permitted by the Standard;
13 programs checking for features defined for each implementation;
3 programs checking for extensions.

B.A. Wichmann
Z.J. Ciechanowicz, extension 3977
For BSI, J. Hatton-Smooker, telephone 0442-3111

A Better Referencer

By J. Yavner
Money Management Systems Inc.

The program which follows was developed from the Currie/Sale procedure cross-referencer published in *Pascal News #17*. Of course, any programmer who looks at someone else's program thinks he could do a better job, but I think that by almost any standard I succeeded, though it took me much longer than Sale's three days. I have an excuse, however: prior to this one, I had never written a Pascal program; my experience with the language comes solely from the articles, standards proposals, and validation suite which have been published in *PN*.

The program is shorter, simpler, and almost certainly faster: It has half as many source lines as the Currie/Sale version, but the format is different, and the number of statements in only 25% smaller. The HP 1000/2015 compiler generated 4604 words of code and static data (the 1000 is not stack-oriented). The procedure descriptor is 25% smaller; the reference descriptor is 97% smaller. The syntax analyzer is more tolerant: missing semicolons do not faze it. The program needs 29.80 seconds and 1376 words of heap to process itself, 356.80 seconds and 5780 words to process the 103-procedure, 4000-line P4 compiler.

The improvement stems from the use of a different data structure: The Currie/Sale referencer is optimized for programs of virtually infinite size, using trees and stack and rings of procedure descriptors and chains of reference descriptors, which allow the procedure database to grow very large with the program's taking all the memory ever manufactured and all the time till doomsday to process it. This referencer, on the other hand, is optimized for small programs, and uses an array of procedure descriptor pointers whose size is fixed by a constant, a quick-and-dirty replacement sort, and sets of reference descriptor flags (two sets, since the program prints the reference data from both viewpoints, caller and callee). As the program to be processed increases in size, memory use increases quadratically, eventually surpassing the Currie/Sale referencer, which started out higher but rises only linearly. Execution time, I imagine, ought to expand similarly. It might be interesting to determine where the cross-over point is.

The program uses the CASE . . . OTHERWISE construct which many processors don't recognize yet. The solution for this problem is to upgrade the processor! An interim fix is to replace the CASE's with IF . . . ELSEIF constructs.

Optional lines

Those lines which begin with the null comment are not vitally necessary to the program and can be removed without seriously affecting its operation. They serve primarily to handle HP1000 extensions.

Lines 19, 21-24, 49-51, 68-71, 522-526, and 551-564 make use of implementation-dependent intrinsics to print processing time and heap usage information. These lines can of course be replaced with the appropriate code to do the job at the target installation or simply left out — like most statistics, they're not really necessary.

Lines 113-116 ignore compiler directives. HP Pascal/1000 has its directives bounded by dollar signs. The format is like strings or comments, and thus is in the spirit of Pascal, but nonetheless the construct must be handled separately.

Line 1 is a compiler directive (another is on line 71). The default output line width is 128, which causes 132-character lines to wrap around even though there are still empty columns on the page.

Lines 307-308 and 345-346 add the HP1000 intrinsics to the pre-defined procedure table. They can be replaced with the appropriate constants for the target installation or removed to make the program conform to the pure standard. The format is as follows: Each procedure name is followed by a space. A hyphen terminates each constant. The last string ends with a procedure name, space, and hyphen, and is then padded with trailing spaces to **ConstLen**. As many strings as necessary can be added at 307 as long as they have corresponding calls at 345.

The directive "external" is recognized by the referencer. Lines 8, 149, 237-238, and 477 could be modified to allow the it to recognize the target installation's directives. The implementation dependency was included primarily to show how this is to be done. The nature of the dependency is such that it can be left in even if the target doesn't recognize it.

Options

This referencer contains a much more efficient **AddIntrinsics** procedure than does the Currie/Sale version (because intrinsics inclusion is not the default for that referencer, while it is for this one). The feature can be disabled by setting **Intrinsics false**. The procedure itself is quite small and can be left in even if inactivated.

The program is designed to print the reference information from the standpoint both of caller and callee. Naturally, twice as much information takes twice the space and twice the time to print. Either table can be disabled separately by means of **CallsTable** or **CallersTable**. Almost all the code for printing the tables is common. As an aside, when both tables are printed it is sometimes difficult to figure out which direction is represented by which table, even though the table's title says "calls" or "callers." One table contains only a single procedure defined at level 0: the main program. Obviously no procedure can call the main program. Similarly, the other table contains the intrinsic procedures. Obviously they don't make any calls.

The identifiers in the input file are truncated if they are too long to fit into the identifier arrays. The length of these arrays is specified by **IdentLen**. Changing this constant requires corresponding modification to the constants defined on lines 5-8, 83-87, and 210-211.

LineWidth can be set to any appropriate value. Setting it to 80 gives two columns of reference data, which is somewhat hard to read (try setting your terminal width to 2 some time). Setting it to 56 forces the tables to have one reference per line, which is rather vertical but still readable.

MaxProc determines the size of the array of pointers to procedure descriptors, and thus the maximum allowable complexity of the input program and the referencer's static size. If it is set to 64 and the HP-specific intrinsics are removed, there is room for 34 procedures, more than most programs published in *PN* need — more than most programs executable on a processor that can't handle large sets probably need.

StackDepth specifies how many BEGIN/END and CASE/END structured statements imbedded inside the body of a procedure the referencer can handle. Few programmers can create code more complicated than 16 nested structures (the referencer never goes deeper than four), but if desired the stack can be extended easily, since each element in the stack takes only one integer.

Offset is the distance from upper case to lower. The program may be set for EBCDIC by changing this constant to the appropriate value.

One final note: no numbers larger than 32767 are needed by the program. On some processors (such as the HP 1000), significant space can be saved by assigning **MaxInt** to 32767 in the referencer's global constant section.


```
$PREF  T=00003 IS ON CR00034 USING 00014 BLKS R=0000

(*$LINESIZE 132*)
                                  program Pref(input,output);
label 9999;
const
  BlankIdent   = '                ';
  ProcIdent    = 'procedure       ';
  FwdIdent     = 'forward;        ';
  ExtIdent     = 'external;       ';
  Intrinsics   = true { Pre-define intrinsic procedures };
  CallsTable   = true { Print table of references FROM procedures };
  CallersTable = true { Print table of references TO   procedures };
  IdentLen     =   16 { Significance limit for identifiers. };
  LineWidth    =  132 { Determines number of identifiers per line }
                      { ((LineWidth-IdentLen-22) DIV (IdentLen+2)  };
  MaxProc      =   74 { Maximum number of procedures.  This should }
                      { be set to a convenient set size.           };
  StackDepth   =   16 { Maximum block nesting within a procedure };
  Offset       =   32 { Distance from upper- to lower-case };
(*$NoTime      =   11 { RTE return-time-of-day code };
*)
(*)OneWord     = -32768..32767;
(*)InfoRec     = record a/toh/b/initon/c/d/e/f                : OneWord end;
(*)TimeRec     = record millisecs:secs:minutes:hours:days: OneWord end;
(*)
  Whole        = 0..MaxInt;
  StackRange   = 1..StackDepth;
  ErrorTypes   = (NoProgram,Redefinition,TooManyProcs,Misplaced,
                  TooDeep,LostEnds,LostPeriod);

  IdentRange   = 1..IdentLen;
  IdentSet     = set of IdentRange;
  IdentString  = packed array[IdentRange] of Char;
  IdentPtr     = ^IdentString;

  ProcRange    = 1..MaxProc;
  LowProcRange = 0..MaxProc;
  ProcSet      = set of ProcRange;
  ProcDesc     = record
    name       : IdentString;
    nameCases  : IdentSet;
    level      : LowProcRange;
    score      : LowProcRange { 0 : Out} 1 : In} )1 : Occluded });
    defline    : Whole;
    bodyline   : -1..MaxInt;
    calls, callers : ProcSet;
    end;

var
(*)info        : InfoRec;
(*)time        : TimeRec;
(*)osc: all    : Integer;

  line, paren  : Whole;
  alpha,alphadigit : set of Char;

  ident, proc  : IdentString;
  identcases   : IdentSet;
  hexident     : IdentPtr;
  identure     : ((InProcess,Other,Def,Directive,Open,CaseOpen,Close));

  procnum      : ProcRange;
  block        : LowProcRange;
  list         : array[ProcRange] of ProcDesc;
  sortlist     : array[ProcRange] of ProcRange;

  bracket      : StackRange;
  stack        : array[StackRange] of Whole;
(*)
(*)procedure Exec(code:OneWord; var time:TimeRec); external;
(*)
(*)procedure GetInfo GALIAS '@UHBI'S (var info:InfoRec); external;

procedure Read; forward;

procedure ReadIdent;
{ Read next identifier from input, keeping track of parenthesis }
{ and skipping comments, quotations, punctuation marks,        }
{ numbers, and compiler directives.                            }
{@alpha,alphadigit,BlankIdent,ExtIdent,FwdIdent,ident,identcases; }
{ IdentLen,IdentRange,identure,input,Offset,paren,!Read          }

  const
    ProcIdent  = 'procedure       ';
    FuncIdent  = 'function        ';
    BeginIdent = 'begin           ';
    CaseIdent  = 'case            ';
    EndIdent   = 'end             ';
  var
    j  : IdentRange;
    ch : Char;

  procedure SkipDigits;
  { Skip numeric characters }
  {@!Read }

    begin while (input^>='0') AND (input^<='9') do Read end;

  begin { ReadIdent }
    ident      := BlankIdent;
    identcases := [];
    identure   := InProcess;
    repeat
      ch := input^;
      if ch='{' then begin { Parenthesis or comment }
        Read;
        if input^='$' then paren := paren+1 else begin { Comment }
          Read;
          ch:='}';
          end;
        end;
      if ch='}' then paren := paren-1;
      if ch='''' then repeat Read until input^='''';
(*)   if ch='@' then repeat { Compiler directive }
(*)     repeat Read until (input^='#') OR (input^='''');
(*)     if input^='''' then repeat Read until input^='''';
(*)     until input^='#';
      if ch='(' then repeat { Comment }
        while (input^='*') AND (input^()='*') do Read;
        if input^=')' then Read;
        until (input^='*') OR (input^=')');
      if (ch>='0') AND (ch<='9') then begin { Number }
        SkipDigits;
        if input^='.' then begin { Decimal }
          Read;
          SkipDigits;
          end;
        if (input^='E') OR (input^='e') then begin { Exponent }
          Read;
          Read;
          SkipDigits;
          end;
        end
      else if input^ IN alphadigit then begin { Identifier }
        j := 1;
        repeat
          if ident[j]<>' ' then j := j+1;
          ident[j] := input^;
          Read;
          until( NOT (input^ IN alphadigit) OR (j=IdentLen));
        for j := j downto 1 do if ident[j] IN alpha
          then begin { Convert to lower case }
            ident[j]  := chr(ord(ident[j])+Offset);
            identcases := identcases+[j];
            end;
        identure := Other;
        if input^ IN alphadigit
          then repeat Read until( NOT (input^ IN alphadigit)
          else if ((ident=ProcIdent)OR(ident=FuncIdent) then identure:=Def
          else if ((ident=FwdIdent) OR ((ident=ExtIdent)
            then identure:=Directive
          else if ident=BeginIdent then identure:=Open
          else if ident=CaseIdent then identure:=CaseOpen
          else if ident=EndIdent then identure:=Close;
          end
        else if ch()''' then Read;
      until (identure()InProcess);
    end;

  procedure Error(error:ErrorTypes);
```

```
  Print error message and terminate )
(0:9999;abracket,ErrorTypes;line,MaxProc:output,stack,StackDepth )

  begin
    write(output,'****** Error at',line:5,' ******* : ');
    case error of
NoProgram:
      writeln(output,'File does not begin with "program"');
redefinition:
      writeln(output,'Procedure defined twice at same scope');
TooManyProcs:
      writeln(output,'Too many procedures, max',MaxProc:4);
Misplaced:
      writeln(output,'Misplaced reserved word');
TooDeep:
      writeln(output,'Too many nested blocks, max',StackDepth:3);
CostEnds:
      writeln(output,'End-of-file -- missing END''s?');
CostPeriod:
      writeln(output,'Unmatched END''s or missing EOF period');
    end;
    if bracket<>1 then writeln(output,'Unterminated blocks:');
    while bracket>1 do begin {Print line #'s of unmatched BEGIN/CASE's}
      bracket := bracket-1;
      writeln(output,stack[bracket]:10);
    end;
  goto 9999;
  end;
function FormatIdent(var ident:Ident;Strings;
var identcases:IdentSet; IdentPtr;
{ Restore upper-case for printing.  Pointers are used because the }
{ result of a function must be either ordinal or pointer.          }
(Hhear:ident,IdentLen,IdentPtr,IdentRange,IdentSet,IdentString,Offset )

  var j: IdentRange;
  begin
    FormatIdent := hear:ident;
    hear:ident^ := ident;
    for j := 1 to IdentLen do if j IN identcases then
     hear:ident^[j]:=chr(ord(hear:ident^[j])-Offset);
  end;

procedure PrintTable(callstable:Boolean);
{ Formatted output of collected reference data.  If several          }
{ procedures have the same name, they appear in order of definition }
(0:Boolean;FormatIdent,IdentLen,LineWidth;list,output, )
procnum,ProcRange:ProcSet,prog,sortlist                              )

  const
    IdentWidth = 18 { IdentLen+2 ( two spaces before ident) };
    Indent     = 38 { continuation indentation? IdentLen+22 };

  var
    j          : 1..LineWidth;
    proc, ref  : ProcRange;
    refset     : ProcSet;

  begin
    writeln(output);
    writeln(output);
    write(output,'  def   body  level  Table of ');
    if callstable then write(output,'calls')
     else write(output,'callers');
    writeln(output,' for ',prog);
    writeln(output);
    for proc:=1 to procnum do with list[sortlist[proc]]  do
     if callstable AND (calls<>[]) OR NOT callstable AND
      ((callers<>[]) OR (level=0)) then begin { Include each procedure }
     { if it called or was called, but include all user-defineds          }
     { in the table of callers in order to find never-useds               }
      j:=Indent;
      if defline=0 then write(output,' ':12) else begin {Non-intrinsics}
        write(output,defline:5);
        case bodyline of

          write(output,'  none?') {Body of forward procedure not found};

          write(output,' extern');

          write(output,' formal')

        else
          write(output,bodyline:7);
        end;
      write(output,level:7,'   ',FormatIdent(name,namecases),':',j:1);
      if callstable then refset:=calls else refset:=callers;
      for ref:=1 to procnum do if sort(list[ref] IN refset then begin
        if IdentWidth>(LineWidth-j) then begin { No room left on line }
          writeln(output);
          write(output,' ':Indent);
          j:=Indent;
        end;
        with list[sortlist[ref]]^ do write(output,'  ',
         FormatIdent(name,namecases)^);
        j:=j+IdentWidth;
      end;
    writeln(output);
    end;
  end;

function FindProc(var proc: ProcRange): Boolean;
{ Set proc to list[] element that points to the  }
{ ProcDesc for ident. If none, result is false. }
(:Boolean;ident,list,procnum,ProcRange           )

begin if list[1]=NIL then FindProc:=false else begin
  proc:=procnum;
  while ((list[proc]^.name()<>ident) OR (list[proc]^.score=0)) AND
   (proc>1) do proc:=proc-1;
  FindProc:=true;
  with list[proc]^ do if(scope<>1)OR(name()<>ident)then FindProc:=false;
  end; end;

procedure AddProc;
{ Add a ProcDesc for ident }
(:block,:Error,:FindProc;ident,identcases, )
line,list,MaxProc:procnum,ProcRange      )

  var proc:ProcRange;
  begin
    if FindProc(proc) then if list[proc]^.level=block
     then Error(Redefinition);
    if procnum=MaxProc then Error(TooManyProcs);
    if list[1]<>NIL then procnum:=procnum+1;
    new(list[procnum]);
    with list[procnum]^ do begin { Initialize }
      name      := ident;
      namecases := identcases;
      level     := block;
      score     := 1;
      defline   := line;
      bodyline  := 0;
      calls     := [];
      callers   := [];
      end
    end;

procedure AddIntrinsics;
{ Add the pre-defined procedures to the procedure list }
(0:BlankIdent,Ident,identcases,IdentRange,:line )

  const
    ConstLen = 53 { Length of the intrinsics-definition constants };
    Const1   = 'abs arctan chr cos dispose eof eoln exp get ln new o-';
    Const2   = 'dd ord pack page pred put read readln reset rewrite -';
    Const3   = 'round sin sqr sqrt succ trunc unpack write writeln - ';
{}  Const4   = 'append close halt lineposs mark maxpos open overprint-';
{}  Const5   = ' position prompt readdir release seek writedir -     ';
    type
      ConstRange = 1..ConstLen;
      ConString  = packed array[ConstRange] of Char;
    var k:IdentRange;

    procedure AddIntrinsics(names:ConString);
    { Do the real work of the procedure.  Necessary since the intrinsics }
    { definition constant is sectioned.  Inner proc is called for each.  }
    (0:AddProc;BlankIdent,ConString,ConstRange,=ident,k )

      var j:ConstRange;
      begin
        j:=1;
        repeat
          if names[j]=' ' then begin { Add procedure }
            AddProc;
            j      := j+1;
            k      := k-1;
            ident := BlankIdent;
            end
          else if names()='-' then begin { Read next char }
            ident[k] := names[j];
            k        := k+1;
            j        := j+1;
            end;
        until names[j]='-';
      end;

    begin { AddIntrinsics (outer) }
      ident      := BlankIdent;
      identcases := [];
      line       := 0;
      k          := 1;
      AddIntrinsics(Const1);
      AddIntrinsics(Const2);
      AddIntrinsics(Const3);
{}    AddIntrinsics(Const4);
{}    AddIntrinsics(Const5);
      line  := 1
      end

procedure ProcessBlock;
{ Process a procedure, function, or program block }
(0:AddProc;block,Extident,:FindProc;FormatIdent,ident,identcases, )
{ identype,line,list,LowProcRange,output,param,procnum,ProcRange    )

  var
    proc              : ProcRange;
    current,localroot : LowProcRange;

  procedure ScanArguments;
  { Read arguments, checking for score occlusions and formal procs }
  (0:AddProc;block,!Error,!FindProc,!FormatIdent,ident,identcases, )
  { identype,line,list,output,param,error,!ReadIdent                 )

    begin { ScanArguments }
      param:=0 { Should be anyway, but make sure };
      ReadIdent;
      while param>0 do begin { Inside argument list }
        if identype=Other then if FindProc(proc)
         then list[proc]^.score:=block
         else else if identype()Def then Error(Misplaced)
         else begin { Formal procedure/function }
          ReadIdent;
          writeln(output,line:5,'  '(block+2,
           FormatIdent(ident,identcases)^));
          writeln(output,line:5,'  '(block+1)+2,'forael'));
          AddProc;
          list[procnum]^.bodyline:=MaxInt;
          end;
        ReadIdent;
        if param>1 then repeat ReadIdent until param(2)
        end;
      end;

procedure ScanDefs;
{ Read definitions, checking for score occlusions and local procs }
(0:block,!FindProc,!FormatIdent,ident,identcases,identype,line, )
{ elist,output,proc,!ProcessBlock,!ReadIdent                     )

  begin while (identype()Open) AND (identype()Directive) do begin
    if identype=Other then if FindProc(proc)
     then list[proc]^.score:=block
     else else if identype=Def then begin { Local procedure }
      ReadIdent;
      writeln(output,line:5,'  '(block+2,
       FormatIdent(ident,identcases)^))
        ProcessBlock
        end
      ReadIdent
      end
    end;

  procedure ScanBody;
  { Check body for references to procedures }
  (0:bracket,current,!Error,!FindProc,=identype,input,line,list, )
  { proc,!Read,!ReadIdent                                        )

    procedure Push;
    { Stack a 'begin' or 'case' statement-bracket line-number }
    (0:=bracket,!Error,line,=stack,StackDepth )

      begin
        stack[bracket]:=line;
        if bracket=StackDepth then Error(TooDeep);
        bracket:=bracket+1;
        end;

    procedure Pop;
    (0:block,=bracket,!Error,input )

      begin if (input^=')') AND ((bracket)2) OR (block)1))
       then Error(LostEnds) else bracket:=bracket-1 end;

    begin { ScanBody }
      list[current]^.bodyline:=line;
      Push;
      repeat
        ReadIdent;
        case identure of
Def,Directive:
          Error(Misplaced);
Open:CaseOpen:
          Push;
Close:
          Pop;
Other:
          begin { Possible reference or assignment to a function }
            while input^=' ' do Read;
            if input^='(' then begin { '(' possible }
              Read;
              if input^=':' then identype:=Def { Assignment }
              end;
            if identype=Other then if FindProc(proc) then begin { Ref }
              with list[current]^ do calls:=calls+[proc];
              with list[proc]^ do callers:=callers+[current];
              end;
            end;
        end;
      until bracket=1;
    end;

procedure DeleteDefs;
{ Set local procedures out-of-score and re-instate occluded ones }
(0:block,list,localroot,procnum,ProcRange )

  var proc:ProcRange;
  begin
    if localroot<procnum then for proc:=localroot+1 to procnum do
     list[proc]^.score:=0;
    for proc:=localroot downto 1 do
     if list[proc]^.score=block then list[proc]^.score:=1;
    end;

begin { ProcessBlock }
  current := 0;
  if FindProc(proc) then with list[proc]^ do
   if (level=block) AND (bodyline=0)
    then current:=proc { Body for a forward-declared procedure }
  if current=0 then begin { Add new procedure }
    AddProc;
    current:=procnum;
    end;
  localroot:=procnum;
  block:=block+1;
  ScanArguments;
  ScanDefs;
  writeln(output,line:5,'  '(block+2,FormatIdent(ident,identcases)^));
  if identype=Open then ScanBody
   else if ident=ExtIdent then list[current]^.bodyline:=-1;
  DeleteDefs;
  block:=block-1;
  end;

procedure Sort;
{ Since there are so few procedures to sort, there }
{ is no need for a complicated algorithm           }
(0:list,procnum,ProcRange,=sortlist )

  var
    proc,j,k : ProcRange;
    status   : (InProcess,Finished);

  begin
    sortlist[1]:=1;
    for proc:=2 to procnum do begin
      k:=proc DIV 2;
      status:=InProcess;
      with list[proc]^ do
       if list[sortlist[k]]^.name)name then repeat
         if k=1 then status:=Finished
          else if list[sortlist[k-1]]^.name)name then k:=k-1
          else status:=Finished
         until status=Finished
       else repeat
         if k=proc then status:=Finished
          else if list[sortlist[k]]^.name(=name then k:=k+1
          else status:=Finished
         until status=Finished;
      for j:=proc downto k+1 do sortlist[j]:=sortlist[j-1];
      sortlist[k]:=proc;
      end;
    end;

procedure Read;
{ Check each char from input for end-of-line }
(0:input,=line )

  begin
    get(input);
    if eoln(input) then line:=line+1;
    end;

begin { Pref }
(:Exec(0:Time,time));
(:with time do begin { Save start time }
(:  sec := days*86400+hours*3600+minutes*60+secs;
(:  all := all1+secs;
(:  end;
  alpha       := ['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'];
  alphadigit := alpha+['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','0','1','9'];

  listC1]     := NIL;
  param       := 0;
  eoln        := 0;
  bracket     := 1;
  line        := 1;
  procnum     := 1;
  new(hear:ident);
  if Intrinsics then AddIntrinsics;
  ReadIdent;
  if ident()ProgIdent then Error(NoProgram);
  ReadIdent;
  prog:=FormatIdent(ident,identcases)^;
  writeln(output,' line  Table of definitions for ',prog);
  writeln(output);
  ProcessBlock { Phase 1 - Process input };
  if input^()'.' then Error(LostPeriod);
  Sort { Phase 2 };
  If CallsTable  then PrintTable(true)  { Phase 3A };
  If CallersTable then PrintTable(false) { Phase 3B };
(:writeln(output));
(:GetInfo(info);
(:Exec(0:Time,time));
(:with info,time do begin { Print statistics }
(:  sec := days*86400+hours*3600+minutes*60+secs-sec;
(:  all := all1+secs-all;
(:  if all<0 then begin { Correct for borrow from all1+seconds }
(:    all := all+100;
(:    sec := sec-1;
(:    end;
(:  write(output,'Mean = ',sniton-ton:1,' words.  Time = ',sec:1,'.',
(:  if all<10 then write(output,'0');
(:  writeln(output,all:1,' seconds.');
(:  end;
  page(output);
  9999:
    end.
```


Dear Rich:

The software tools section of Pascal News is extremely useful. We have implemented Prose on the HP 3000 and we enjoy using Prose to do our text formatting.

This letter includes one enhancement to Prose and one bug-fix. The enhancement provides a new terminal-type: DIABLO. This terminal-type provides for proportional spacing on DIABLO terminals. The changes are as follows:

Lines 167 to 173 become:

```
    { THE FOLLOWING ARE NOT DIRECTIVES, BUT IT IS CONVENIENT
    { TO INCLUDE THEM IN THIS TABLE.

            AST,            { ASCII TERMINAL              }
            LPT,            { LINE PRINTER                }
            AJT,            { ANDERSON/JACOBSON TERMINAL  }
            DIA,            { DIABLO TERMINAL             }
            ILT);           { ILLEGAL                     }
```

Lines 789 to 793 become:

```
          CASE TERMINALTYPE OF
            AJT,
            DIA,
            AST: WRITE1(CR);
            LPT: BEGIN
                   WRITELN(OUTPUT);
                   CARRIAGECONTROL:=PLUS
                 END
          END
```

Lines 824 to 825 become:

```
            END { IF TERMINALTYPE = AJT }
          ELSE
          IF TERMINALTYPE = DIA THEN
            BEGIN
              X2 := 0;
              FOR X1 := 1 TO LEN DO
```

```
            WITH STR[X] DO
              IF C <> BLANK THEN
                BEGIN
                  IF X2 <> 0 THEN
                    BEGIN
                      IF (X2 MOD CHARWIDTH = 0) THEN
                        FOR X3 := 1 TO (X2 DIV CHARWIDTH) DO
                          WRITE(BLANK)
                      ELSE
                        BEGIN
                          FOR X3 := 1 TO (X2 DIV CHARWIDTH) DO
                            WRITE(BLANK);
                          X2 := X2 MOD CHARWIDTH;
                          WRITE(ESC);
                          WRITE(THREE);
                          FOR X3 := 1 TO X2 DO
                            WRITE(BLANK);
                          WRITE(ESC);
                          WRITE(FOUR);
                        END
                    END;
                  X2 := 0;
                  WRITE(C)
                END
              ELSE  X2 := X2 + NB1
          END
        ELSE
        FOR X1 := 1 TO LEN DO

Lines 1852 to 1860 become:

        AJT,
        DIA: BEGIN
               WHILE INCHAR = BLANK DO
                 NEXTCH;
               CHARWIDTH := NUMBER(10, -1, 0, INFINITY, 1013);
               IF NOT (CHARWIDTH IN [10, 12]) THEN
                 BEGIN
                   ERROR(1013);
                   CHARWIDTH := 10
                 END;
               IF (TERMINALTYPE = DIA) AND (CHARWIDTH = 12) THEN
                 BEGIN
                   WRITE(ESC);     {Write out the HMI}
                   WRITE(US);
                   WRITE(FF);
                 END;
               CHARWIDTH := 60 DIV CHARWIDTH;
               OUTLINE[1].NB1 := LEFTMARGIN * CHARWIDTH
             END

Lines 3439 to 3440 become:

  IF ERRORS THEN WRITELN(' PROSE ERRORS DETECTED.');
  IF (TERMINALTYPE = DIA) AND (CHARWIDTH = 5) THEN
    BEGIN    {RESET PITCH}
      WRITE(ESC);
      WRITE(S);
    END
  END.  { PROSE }
```

The version of Prose published in PN # 15 contains a bug concerning index entries. If an index entry is underlined, Prose starts referencing the NIL pointer. The problem is that the function UPPER returns an incorrect value for underlined characters. A new UPPER function is introduced in the SORT procedure.

```
Lines 2169 to 2170 become:

        X1          : INTEGER;  { GENERAL INDEX VARIABLE       }

{           UPPER - SPECIAL VERSION OF UPPER.  DOES NOT RETURN
 *                  UNDERLINED CHARACTERS.
 *
 *           PARM CH = CHARACTER TO CONVERT TO UPPER CASE.
 }

      FUNCTION UPPER( CH : ASCIIX) : ASCIIX;
      BEGIN { UPPER }
        IF ODD(CH DIV 128) THEN
          CH := CH - 128;
        IF CLASS(CH).LETTER THEN
          IF CH >= SMALLA THEN
            UPPER := CH - 32
          ELSE
            UPPER := CH
        ELSE
          UPPER := CH;
      END {UPPER};


      BEGIN  { SORT }
```

I encourage all Prose users to send their changes to Pascal News. With such an excellent tool it would be unfortunate if widely varying versions were to start appearing.

Yours truly,
David J. Greer

The Use of Generic Capsules with the University of Minnesota Pascal 6000 Compiler

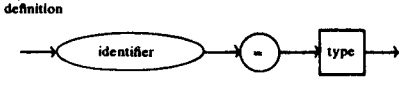
by Frank L. Friedman
Alessio Giacomucci
Carol A. Ginsberg
Anita Girton
Temple University

I. INTRODUCTION

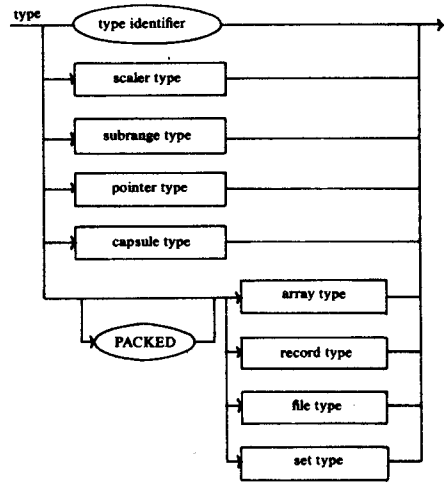
This document contains a description of a data type abstraction facility, a *capsule*, that has been implemented as an extension to the University of Minnesota Pascal 6000 Series compiler. The facility provides an encapsulation that establishes a static scope of identifiers with controlled visability. Data objects and a set of operations on these objects may be enclosed. The document is intended to provide sufficient information for those who wish to use the general capsule facility and library. A more complete description of capsules may be found in the paper "Capsules: A Data Abstraction Facility for Pascal," CIS-TR 81-01, Temple University C & IN SC Department Technical Report.

II. WHAT IS A CAPSULE?

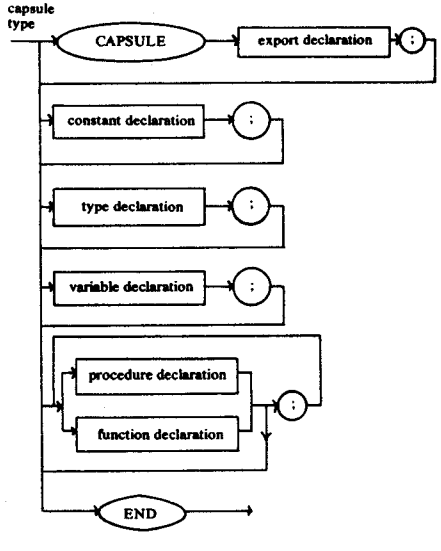
A capsule is an additional Pascal type which is syntactically similar in structure to the Pascal *record*. The syntax diagrams for the Pascal type definition (with the capsule added) may be specified as



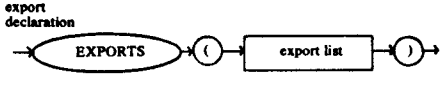
Department of Computer and Information Sciences, Computer Users Document 81-01, February, 1981, Rev. 1, September, 1981, Rev. 2, December, 1981



The capsule type is defined by the diagram



with the export declaration defined as



The export list is a list of variable, procedure and function identifiers which may be referenced outside the scope of the capsule. All protection of the data objects encapsulated in the capsule is provided at compile time. Thus, if capstype is a capsule, and the variable X is declared to be of type capstype, then all external references to identifiers, id, appearing in the export list for capstype must be of the form

X$id

Exported variables are read only, and identifiers not appearing in the export list may not be referenced outside the scope of the capsule. There is no explicit import facility, such as provided in Modula and Euclid.

The Pascal scope rules for capsules are the same as the rules for all other Pascal objects. Only a single copy of the operations (procedures and functions) defined within a capsule is created, regardless of the number of variables declared to be of the capsule type. When a procedure (or function) containing the declaration of a capsule-type variable is called and the variable declaration is elaborated, the capsule's global variables are placed on the runtime stack as a record. This record remains on the stack as long as the called procedure (function) remains active. Operations on the abstract objects are thus performed via calls of the appropriate capsule procedures or functions.

An example of a capsule in parameterized (generic) form is shown in Figure 1. An illustration of the use of this capsule is shown in Figure 1.

```
                {A non-recursive expression parser}

                type

A.                  $G ('capstk'/'capsall', charstack20,20,char);
                    .
                    .
                var
                    stack:  charstack20;
B.                  .
                    .
C.              begin     {initialize} stack$init;
                    .
                    .
                {   stack$push (cursym);
D.                  .
                    .
                    stack$pop (rightoperand);
                    stack$pop (operator);
                    stack$pop (leftoperand)
                    .
                    .
                end {parser};
```

Figure 1:

Use of a simple stack capsule

```
E.  capstk

F.  (pname, psize, ptype)        {list of capsule parameters}

    pname = capsule

    {stack capsule definition (in generic form)
     *
     *
     *  parameters:
     *  pname  - name of capsule
     *  psize  - number of elements in the stack
     *  ptype  - base type of stack array
     }
```