PASCAL USERS GROUP

Pascal News

NUMBER 21

COMMUNICATIONS ABOUT THE PROGRAMMING LANGUAGE PASCAL BY PASCALERS

APRIL, 1981

If this isn't APRIL...

does that mean we're late ?

POLICY: PASCAL NEWS (15-Sep-80)

* 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!

  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 that we can do."

* Pascal News is produced 3 or 4 times during a year; usually in March, June, September, and December.

* 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 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 - 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.

  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.

- - - - - - ALL-PURPOSE COUPON - - - - - - (1-Apr-81)

Pascal Users Group
P.O. Box 4406
Allentown, Pa. 18170-4406 USA

Note

- We will not accept purchace orders.
- Make checks payable to: "Pascal Users Group", drawn on a U.S. bank in U.S. dollars.
- See the Policy section on the reverse side alternate address if you are located in the Australasian Region.
- Note the discounts below, for multi-year subscription and renewal.
- The U. S. Postal Service does not forward Pascal News.

- -

		USA	Europe	Aust.
[ ] Enter me as a new member for:	[ ] 1 year	$10.	$14.	A$ 8.
	[ ] 2 years	$18.	$25.	A$ 15.
[ ] Renew my subscription for:	[ ] 3 years	$25.	#35.	A$ 20.

[ ] Send Back Issue(s) _____

[ ] My new 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: A$ _____
$ _____.____
CHECK no. _____

NAME _____

ADDRESS _____

_____

_____

PHONE _____

COMPUTER _____

DATE _____

JOINING PASCAL USERS 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 a year: January 1 to December 31, 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), and European Region (Europe, North Africa, Western and Central Asia): Join through PUGUSA

- Australasian Region (Australia, East Asia - incl. Japan): PUG(AUS). Send $A10.00 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 x435

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 Region must join through their regional representative. People in other places please join through PUG(USA).

RENEWING?

- Please renew early (before November 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 a year 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.
- Issues 9 .. 12 (September, 1977 - June, 1978) are available from PUG(USA) all for $15.00 and from PUG(AUS) all for $A15.00
- Issues 13 .. 16 are available from PUG(AUS) all for $A15.00; and from PUG(USA) all for $15.00.
- Extra single copies of new issues (current academic year) are: $5.00 each - PUG(USA); and $A5.00 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. Please send material single-spaced and in camera-ready (use a dark ribbon and lines 18.5 cm. wide) form.
- All letters will be printed unless they contain a request to the contrary.

APPLICATION FOR LICENSE TO USE VALIDATION SUITE FOR PASCAL

Name and address of requestor:
(Company name if requestor is a company) _____

Phone Number: _____

Name and address to which information should be addressed (Write "as above" if the same) _____

Signature of requestor: _____

Date: _____

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

a) The Validation Suite is recognized as being the copyrighted, proprietary property of R. A. Freak and A.H.J. Sale, and

b) The requestor will not distribute or otherwise make available machine-readable copies of the Validation Suite, modified or unmodified, to any third party without written permission of the copyright holders.

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 to make available the listings of the results of compilation and execution of the programs to third parties in the course of the above activities. In such documents, reference shall be made to the original copyright notice and its source.

✓ Distribution charge: $50.00

✗ Make checks payable to ANPA/RI in US dollars drawn on a US bank. Remittance must accompany application.

Mail request to:

ANPA/RI . .
P.O. Box 598
Easton, Pa. 18042
USA
Attn: R.J. Cichelli

Source Code Delivery Medium Specification:
9-track, 800 bpi, NRZI, Odd Parity, 600' Magnetic Tape

( ) ANSI-Standard

a) Select character code set:
( ) ASCII ( ) EBCDIC

b) Each logical record is an 80 character card image.
Select block size in logical records per block.
( ) 40 ( ) 20 ( ) 10

( ) Special DEC System Alternates:
( ) RSX-IAS PIP Format
( ) DOS-RSTS FLX Format

Office use only

Signed _____
Date _____

Richard J. Cichelli
On behalf of A.H.J. Sale & R.A. Freak

Index

Contributors to this issue (#21) were:

EDITOR	Rick Shaw
Here & There	John Eisenberg
Books & Articles	Rich Stevens
Applications	Rich Cichelli, Andy Mickel
Standards	Jim Miner, Tony Addyman
Implementation Notes	Bob Dietrich, Greg Marshall
Administration	Moe Ford, Jennie Sinclair

Editor's Contribution

NEW ADDRESS

Yes, in my continued effort to bring you better service, (read this as: I can not do all the work effectively!) I have found someone else (read: sucker) to take over the PUG mailing list. I am sure that this will increase the satisfaction level for this task 100%. this will take a great load off of my back and allow me to devote all of my time to editing and publishing Pascal News.

LATE

I thought April first (April Fools Day) was an appropriate target date for this issue of Pascal News! I apologize for the tardiness, but my work (I have a real job that pays the bills) and the many pressing problems and issues of PN got in the way. I had to solve the PUG Europe problem, and try to gather as much as I could concerning the final vote on the ISO standard.

FUTURE OF PUG IN EUROPE

It took me more than a few months to correct the festering problems in Europe surrounding Pascal News. The previous coordinator was sinking under the mire of ever increasing job responsibilities as well as the editorship of clearly the best journal dealing with practical software implementation. (SP&E) As a result, the european region suffered from lack of attention. This is over! PUG cares. Please send your "job well done's" to David in Southampton, and send your complaints to PUGUSA. We will be handling all but the Australasia Region from the US. Please read the new APC carefully for policy and price changes. We will be mailing by surface mail to the UK and Europe, but I have been assured by the USPS that it should take no more than a month. I have been asked if I would mail by air for an extra surcharge. The answer has to be no, at this time. PUG can just not afford the special processing and handling that this would be required for two different types of mail. Sorry!

STANDARDS

Another delay was the standards effort. There is so much going on in the standards arena that we just could not afford to miss it. I think it was worth it. Over half of this issue is devoted to the vote on the ISO standard for Pascal (7185). Jim Miner has done another fine job.

THIS ISSUE

Now the good news! We have another jam packed issue. I think you will recognize out book reviewer this issue. He is an "occasional" contributor to PN. And I hope you will get a chuckle from our "sister" publication PUG PRESS. Andy Mickel brings this little gem to us. The other HERE and THERE article is a real puzzle. It came to me just as you see it!?

The application for this issue was so good I could not miss publishing it. It is a Pascal to EM1 pseudo code compiler by Andrew Tanenbaum. Its a real beauty. But it was sooooo big I could not publish it all ... yet. This issue contains the definition of the assembler language that is output and also an interpreter which serves as the EM1 machine definition. Issue 22 will contain the program text for the EM1 Pascal compiler. I hope everyone reviews the documentation and the code, even if they do not need the compiler. It is a fine example of elegant design and implementation using the language Pascal. Also included in the APPLICATIONS section is an article by Jeff Pepper om the implementation of extended precision integer arithmetic. A fine job.

The ARTICLES section contains a thought provoking extension to the read/write subroutines by David Rowland. Lets hear a response from the members. And finally Maragret Kulos has contributed a very comprehensive article comparing OMSI-1 Pascal and The Swedish Pascal compiler. There is a great deal of interest in these two compilers for the PDP-11. I hope this provides some answers.

All in all, a great issue. More to come on EM1 in issue #22.

Hope you like it!

Rick

****

Here and There With Pascal

BOOK REVIEW

The Pascal Handbook by Jacques Tiberghien
500pp, 270 Illustrations, SYBEX, Berkeley
(1980) $US14.95 (paper edition only),
ISBN 0-89588-053-9.

Overview

This is not a Pascal textbook; it is something very different. Perhaps the most succint description is that it is a Pascal *lexicon*: a sort of all-purpose reference manual. It is organized around entries keyed by an appropriate Pascal word (eg if, scope, writeln) arranged in alphabetical order. Each entry takes up one or more whole pages, and the standard sub-headings are SYNTAX, DESCRIPTION, IMPLEMENTATION-DEPENDENT FEATURES and EXAMPLE. The relevance of the entry to Standard Pascal and a number of particular implementations (HP1000, CDC, OMSI-1, Pascal/Z and UCSD Pascal) is encoded into the entry.

Thus the book is meant to be used as a dictionary to look up difficult points or to find out what some usage in a program you have received really means. As such, it follows a lot of reference manuals which are similarly structured (eg the B6700/7700 Pascal Reference Manual).

However, since Pascal is a small language with not very many things needing to be remembered, it needs to be asked why a lexicon of 500 pages is needed? Examination of the book indicates that its main purpose seems to be to document extensions and differences between implementations. Thus, since its topic is the union of all the quirks of 5 implementations, it has grown to this rather large size.

Target and reality

So much for the target; how does the book match up to it in reality? The answer seems to be that it does a reasonably good job of documenting what exists, but that it does not measure up to the very exacting standards that such an ambitious project warrants. The standard of accuracy against which a dictionary is judged is much higher than that appropriate for textbooks, in which a few lapses can be tolerated or justified on the grounds that pedantic accuracy would impede learning.

The slips in the book are far too numerous to detail (a list is being sent to the author), and a few examples will have to suffice. Dipping into the entry for the reserved word for is probably the richest source of examples. Faults which should be mentioned are:

(1) An "equivalent flow-chart" is given. The sense of defining a high-level construct such as while in flow-chart terms is questionable at the best of times, but for the complex for-statement it is extremely unfortunate in that it might make people think the flow-chart is right. It isn't.
(2) The prohibition on changing the value of the count variable is not mentioned.
(3) The limitations on what a count variable can be (only a local simple variable) are not mentioned.
(4) The correct restriction of the HP-1000 implementation is considered to be an implementation-dependent feature, whereas the corresponding flaw in the J&W/CDC implementation is not mentioned.
(5) The failure of many of these implementations to enforce the requirements of the for-statement is not mentioned; indeed for four implementations the entry is *None known* for implementation-dependent features.
(6) The possibility of the statement failing to terminate (incorrectly) for some limit values in the OMSI and UCSD implementations is not documented.
(7) The statement is made that *The A and B parameters may not be modified by the statement in the loop*. This is simply incorrect, though it is true to say that the loop limits are determined on entry to the loop.

Perhaps this is the worst case to show, but a few more examples will suffice to show that the problem is not isolated. The syntax for MARK shows that this non-standard procedure takes a parameter which is an integer expression. MAXINT is incorrectly described as determining the positive limit of representable integers (which it may be only coincidentally). The syntax for CASE statements is incorrect. And so on.

General issues

There are two major deficiencies in this book which deserve comment. First is the lack of formal definitions, and indeed the appearance of only a few English descriptions that resemble the actual requirements of Pascal. The author claims to be talking about Pascal (presumably the standard variety) as well as the others, but there is simply no basis for comparison if the reader cannot find out what sets, for example, are really supposed to be.

The second is the mystifying omission of any reference to the Pascal Validation effort. If one of the purposes of the book is to aid programmers who wish to write portable Pascal programs, then it is difficult to understand why the author did not carry out validation tests on the five compilers he regards as important, and print the results in a second section of the book. It would have added significantly to the value of the book as a reference.

Minor issues

Regrettably, once again it is necessary to point out that capitals were designed for carving into stone, not for ease of reading. This book perpetuates the habit of printing programs in capitals, with consequent loss of legibility.

It is difficult to deduce the author's criteria for choosing which topics to omit or include. To illustrate this, note that the UNIT feature of USCD Pascal, together with the corresponding USES, INTERFACE and IMPLEMENTATION reserved words, is not treated in the book, apart from a mention, despite their undoubted importance in use. On the other hand, such trivia as a pre-defined function EXP10 in OMSI Pascal takes up 2 pages.

Directing another comment to the publisher rather than the author, one wonders why the tremendous amount of white space in the book was tolerated. A little care in layout (perhaps two entries per page; perhaps denser printing) would have halved the number of pages, and perhaps reduced the price.

Summary view

Despite the criticisms made above, I believe the book would be useful to programmers who have to cope with Pascal programs which were developed on different systems or in different dialects. The level of detail and accuracy of information is not as high as it could be, but nevertheless the book has no competitors.

I doubt that it will be of much use to programmers learning Pascal, still less beginners at programming, because it is too difficult to see what is really Pascal and what is "extension". And of course, dictionaries are simply not meant to be read through.

A.H.J.Sale

BOOK REVIEW

INTRODUCTION TO PASCAL

– including UCSD Pascal

by Rodnay Zaks
320pp, 100 illustrations
Sybex, Berkeley (1980) US$12.95 (Paper Edition only)
ISBN 0-89588-050-4

Reviewed by A.H.J.Sale, Sandy Bay, Tasmania.

Overview

On receiving a book which proclaims that it will teach you a programming language, I conceive that most reviewers will groan and wonder what new there is to say. The more so if the language is a popular one, such as BASIC, Fortran, COBOL, or Pascal. For many educational book writers are plagiarists, and after the fifth to tenth version of the same ideas, my eyes get weary and the text fuzzy...

To start with, then, it is a pleasure to be able to write that Rodnay Zaks book is somewhat different from the run-of-the-mill Pascal books. Firstly it has a definite target readership, and is addressed to them. Dr Zaks' book is well-suited to microcomputer enthusiasts and programmers who want to learn a bit about Pascal but have no immediate intention of using it professionally. The exposition is gentle, fairly easy to read, and liberally interlaced with reading examples.

To enhance its value to such readers, Dr Zaks has decided to include material on one popular variant of Pascal in the microcomputer field: UCSD Pascal. This is interspersed throughout the book in clearly labelled sub-sections.

Scoondly, the book has a good collection of examples, and they are not exactly the same examples you find in other textbooks! Learning a language is always easier if you can read it (and read a lot of it), since then you discover samplers (or templates) that you can modify to your own purposes, and thus gradually discover typical programming paradigms of that language.

The presentation is traditional, and there are no surprises. The chapter headings are: Basic Concepts, Programming in Pascal, Scalar Types and Operators, Expressions and Statements, Input and Output, Control Structures, Procedures and Functions, Data Types, Arrays, Records and Variants, Files, Sets, Pointers and Lists, UCSD and Other Pascals, Program Development (15 in all) followed by 12 Appendices including answers to selected exercises.

Shortcomings

In my opinion, the book is not likely to be widely used as a text in tertiary courses, for several reasons. Most importantly, it is very light on the concepts of Pascal and Dr Zaks treats of the language simply as another Fortran or BASIC. Instructors trying to get across the important advances in knowledge about computing will not forgive the lack, whereas readers using it as a self-tutorial almost certainly wouldn't notice the deficiency. Less important, but still relevant, is the typical American verbosity in this kind of book.

To illustrate the conceptual treatment, observe that 6 pages (pp135-140) deal with enumerated types and subranges, and 11 pages (pp247-257) for sets. Other data structuring methods seem to fare better, but this appearance disappears on close examination. For example the array chapter contains 39 pages, but 4 pages are devoted to a matrix addition program, 16 pages to a sorting program, and 8 pages to UCSD features (including UCSD strings which are not arrays at all!), leaving 11 pages of discussion of the syntax and semantics of arrays. The low-level obsession with flow of control is very obvious in this book.

A reviewer cannot pretend to check every program and statement in a book such as this, but I was pleased to note few errors or half-truths in "Introduction to Pascal". Notable amongst the omissions, however, are references to the axiomatic definition of Pascal (surely one of the most important sources!) and to the draft ISO Standard for Pascal. These omissions seem to be related to the book's orientation towards small computers and relatively naive programmers.

In spite of the great care put into this book (its technical presentation is excellent except for the blunder of printing program text in capitals), I had to come to the conclusion that the inclusion of UCSD Pascal in it is a mistake. The book is predominantly about "Standard Pascal", and purchasers who hope to learn something about UCSD Pascal that is not in the UCSD and SofTech manuals will be disappointed. It seems that the UCSD material acts as textual clutter, even if its inclusion on the cover sells more copies.

Summary

"Introduction to Pascal" by Dr Rodnay Zaks is a useful soft-cover book that will probably be useful to people trying to learn Pascal by themselves, due to the many examples. However, it will lead them up to the point of programming using Pascal, but thinking in traditional ways. Many of the insights and productivity improvements will require extensive further experience, but perhaps that is inevitable.

* Pug Press *

Volume One Issue Three March 1980

Publisher: Maryanne Johnson (612)-474-7167
510 Wheeler Drive Excelsior, Minnesota 55331
Editor: Patti Sue Selseth

Even with all the snow on the ground, SPRING IS IN THE AIR!!!! This is a good time to remember to bring your dog's shots up to date and don't forget about heartworm.

One of Marianne's Pug Family has passed away in early February. Helen Landon had only had her PUGS for 2½ years, but she truly loved them. Her love for all animals was a driving force in her life, and she will be missed. The family has requested memorials to Pet Haven or American Cancer Society.

* Congradulations - On Your Newly Aquired PUGS

Tracy Cunningham has a new little girl PUG named Miss Josie Posie Penelope. The day before Christmas she was brought home at the tender age of 2½ weeks. (This should be a reminder that not all breeders are as concerned for the dog's welfare as they should be. There is no excuse for selling a dog at this age for monatary gain. Remind people who are looking for puppies that they should be eating from a dish, and should be able to get along without their Mama and litter mates before they are taken home.)

Mr. and Mrs. Don Coen of South St. Paul are soon to be getting a new baby boy PUG. They recently lost a 13 year old PUG.

Mr. and Mrs. Don Donaldson of River Falls, Wisc. became owners of a male PUG at Christmas time. They bought him from Rachel Fishcher; he was at the Pug Party last fall as a puppy.

Mr. and Mrs. Joe Jenareo of Minot brought home a new female PUG in December. They have an eight year old male and are also looking for another male.

The John Kerschner Family recently bought an eight month old PUG puppy from Dorothy Justad.

* A PUG Name Contest

The John Healy Family would like to know some of the names that have been given to the pugs. So we thought it would be fun to have a "PUG NAME CONTEST." The contest will be based on the registered and/or call names our PUG people have named their PUGS (past and present). Some of the catagories will be: most unusual, most beautiful, most interesting, most common, and most humerous. To enter the contest, please write or call Maryanne before June 1, 1980. All entrants will be mentioned in the next newsletter.

* Have You Heard the Latest???

We have it on good authority that Pandy Wenz has visited Chipper Justad at his home. Early May will tell the tail!!!!

* Birth Announcements

Dorothy Justad is proud to announce the arrival of:

Woodcrofts Foster Fordyce arrived February 12th (the one and only)

Sire: AKC & CKC ptd in Bermuda Ch. Sheffields Shortening Bread (better known as Chipper)
Dam: Sugar Plum Jen I

* Want Ads

WANTED - Small PUG Stud to breed with the Classiest Bitch in Town. Stud must be experienced yet gentle, loving, and discreet. Contact Ron or Marlys Hampe (612)-890-4141

John G. Waltz; 184 Amherst, St. Paul 55105; is the manager at Sherwoc Pet in St. Paul. He would like a male pet PUG at a reasonable price.

Eunice Thorson; 536 1st St., Proctor, Mn. 55810; recently lost her fourteen year old PUG. She would like another girl puppy or older PUG.

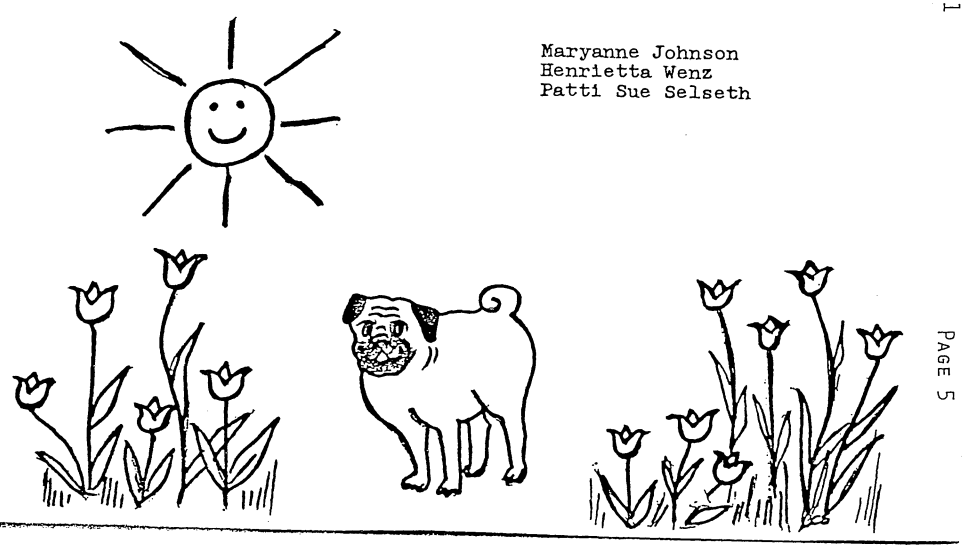
* Thank You - Dorothy

Our thanks to Dorothy Justad for the wonderful article on getting started in show biz. We know it will be useful to those of you interested in showing PUGS.

If you have any PUG news that you would like to share with fellow "PUG PEOPLE" please let us know. Deadline for the next newsletter is June 1, 1980. Just call or write Maryanne, and we'll get your news in the next issue of PUG PRESS.

HAVE A HAPPY SPRING !!!!!!!

Maryanne Johnson
Henrietta Wenz
Patti Sue Selseth



Dear Newsletter enthusiast, the following is a list of subjects that are likely to be examined in the upcoming newsletters. If you feel like it, please respond to any of the subject matter, adding suggestions, visions, or other comments. Bits of any incoming communication are likely to be recycled into the newsletter at some date. This being the first newsletters, the form may change from issue to issue, but my idea initially is to have each letter be a theme examining some proponent of the hypothetical floating sea city, of which we can all be a part.

a. the spiral method of accretion

b acquiring the necessary elements off the land: going into the recycling aspects of the project, recycling of cars, refrigerators, machines for the conducive materials, and also papers and (liquified) plant matter, for the papier mache structures.

c. A deeper tripping out on paper machait: how it can be used to invoke peoples' minds as to the process of accretion, selecting varieties of forms which scintillate. Drawings can be included of terrestrial motifs, walls, time capsules, zoomorphic borders of gardens, rises, walkways, spontaneous expressions of color and form.

d. aspects of energy acquisition and usage: Solar, wind models, underwater exploits; shasing-concepts, valuation.

e. Plantlife likely to evolve, and the natures of emergent ecosystem including overlappings, and new symbioses.

f. Food to be grown, produced, specialty items for shipping away, into the land: Pickles, sweets, noted cheeses, pastries, modes of eating; availability of different substances.

g. Separation of thirds of spaces: industrial/mercantile/co-operative; common/state-owned; and home spaces, privately ruled and operated.

h. Varieties of social forms, explorations of likely traditions to be fused for propulsion into pyremusical ambidextrously mobile batteries. Cultures to be examined including refugees, aliens, star-struck, dropped out, mutating, change-oriented.

i. Blasting of the closed-ended systems, reiterating the expansive potentials inherent in futuristic thinking: an invitation to recent explosions.

j. The inner workings and displayed aspects of the water system in the structure. Designs for waterfalls, ponds, pools, streams, bathing, plant feeding, recirculation, distillation.

k. Art- and Extrapolitical-aspects of lifestyles emerging on the sea. Options for peoples' expressions in career, craft, vocation, activities;

l. An examination of the effort to create groups of three melting, softing tetras, to meet and merge on the high seas, producing the interior lagoons and flatlands. Also known as triangulation, the tendencies of groups of threes to balance and stability.

m. Diagrammatic explanations of the various levels, including shipping ports, flotation devices, fluxuating shores, and sky-high properties. Proportions of spaces allotted to playgrounds, bycicle loops, orchards, cottages, mist gardens, arboretum/terminal stands, geodescic elevating modules, and sky light sculptures of varying densities will be suggested, examined, detailed.

n. Something to attract transient visitors: vacation playgrounds. There are fantasy worlds open for exploration, and technological and entertainment forms. Also perhaps, casino- and pub-like grottoes, looking out to under the waves; and varieties of sports presentations and activities. Contests, fairs, festivals, holidays, erections, revampings, scribblings.

o. Communication with other life forms, and inviting them along for the journey into spaces high and blue. The idea of having a dolphin embassy, a whale tavern in the sea (growing types of algae for them), platforms and niches to support many sea travellers, and those from the sky.

p. A continuously building mural made by contributions from each visitor in all the media. It will start from some initial point(s) and spread as more and more visitors come, make, and go. (Thanx, Yoko)

q. The idea of "not letting an enemy rise on any level", as Maharishi so aptly puts it. The foreign relations applicable as: Ideologies can be shared as love. Using the platform as a museum, a carousel of multiple nationalities and displays of bifurcate merging, develop events which can be generally supported by nations, groups, and factions. In them independent rovers can sniff around.

r. Examinations of the acoustics, the silent cave-likes, the public, open, airy ampetheaters. Electronic and other forms of communication running along its circuits, and extending from its structure.

s. Visions, ideas for schools, markets, subjects to be taught: seems likely there's to be a concentration of the space studies on board, so examining some of the fields briefly: exo-ecology, low gravity motion, non-terrestrial physics, neurogenetic engineering.

t. Health, wholeness, holiness: attaining it and keeping it, some of the newer medicinal statements have been waiting for somewhere like this to display themselves, and from which to fly.

u. The idea as the project not just an end, a new place, but as another link on the roadway. What then is to come next? What first? What has been encouraging this?

v. The extra-realist art movement, its principles and principals.

w. Tributes to those livers of the past who've sent good vibrations into our present sphere. Catacombs and hillsides.

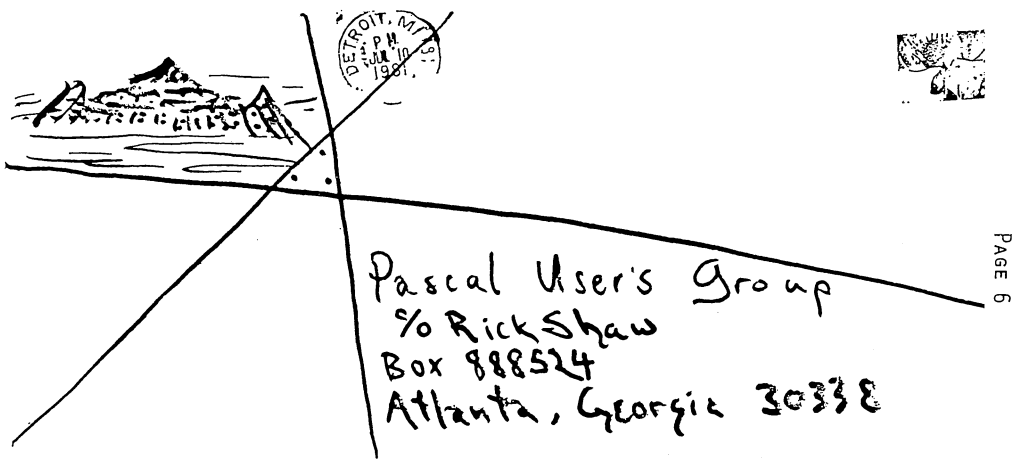
x. The exposition and superimposition of the ideas of nakedness, nudity, nets of reality, and masturbation. Techniques.

y. Proposal for direct access networks to stretch across the land.

z. An animal's or plant's eye view of what we humans have been discussing, sometimes grave, sometimes humorous.

In closing, I would like to add that all flowing waters lead to the sea. Thanks for the initial interest. Direct correspondance to me at: Kevin Switzer, 1534 Ford, Lincoln Park, Michigan 48146.

* **** ** ** * * * ** * * **



Applications

EM-1 ASSEMBLY LANGUAGE

11.1. Introduction

An assembly language program consists of a series of lines, each containing 0 or 1 statements. A machine instruction may not be labeled. In other words, the label field on a machine instruction must be left blank. There are two kinds of labels, instruction and data labels. Labels start in column 1. Instruction labels are unsigned positive integers, and each must appear alone on a line by itself. The scope of an instruction label is its procedure.

The pseudoinstructions CON, ROM, and BSS may be labeled with a 1-8 character data label, the first character of which is a letter, period or underscore, followed by letters, digits, periods and underscores. Only 1 label per line is allowed. The use of the character "." followed by a number (e.g. .40) is recommended for compiler generated programs, since these are considered as a special case and handled more efficiently in compact assembly language (see below).

Each statement may contain an instruction mnemonic or pseudoinstruction. These must begin in column 2 or later (not column 1) and must be followed by a space, tab, semicolon or LF. Everything on the line following a semicolon is taken as a comment.

All constants are decimal unless started with a zero e.g. 0177, in which case they are octal. In CON and ROM pseudoinstructions, floating point numbers are distinguished by the presence of a decimal point or an exponent (indicated by E or e), or both. Double precision (long) integers are followed directly by an L or l.

Also allowed as initializers in CON and ROM are strings. Strings are surrounded by double quotes and may include \xxx, where xxx is a 3-digit octal constant, e.g. CON "hello\012\000". Each string element initializes a single byte. Strings are padded at the end up to a multiple of the word size.

Local labels are referred to as *1, *2, etc. in CON and ROM pseudoinstructions (to distinguish them from constants), but without the asterisk in branch instructions, e.g. BRF 3, not BRF *3.

The notation $procname is used to mean the descriptor number for the procedure with the specified name.

An input file may contain many procedures. A procedure consists of zero or more pseudoinstructions, a PRO statement, a (possibly empty) collection of instructions and pseudoinstructions and finally an END statement. The very last statement on the input file must be EOF. The END directly preceding the EOF may be omitted.

Input to the assembler is in lower case, if available. Upper case is used in this document merely to distinguish key words from the surrounding prose.

11.2. Pseudo instructions

First the notation used for the operands of the pseudo instructions.

<num> = an integer constant
<sym> = an identifier
<arg> = <num> or <sym>
<val> = <arg>, long constant (ending with L or l), real constant, string constant (surrounded by double quotes), procedure number (starting with $) or instruction label (starting with *).
<...>* = zero or more of <...>
<...>+ = one or more of <...>

Four pseudo instructions request global data:

BSS <num>
Reserve <num> bytes, not explicitly initialized. <num> must be a multiple of the word size.

HOL <num>
Idem, but all following absolute global data references will refer to this block.

CON <val>+
Assemble global data words initialized with the <val> constants.

ROM <val>+
Idem, but the initialized data will never be changed.

Three pseudo instructions partition the input into procedures:

PRO <sym>,<num1>,<num2>
Start of procedure. <sym> is the procedure name. <num1> is the number of bytes for arguments. <num2> is 1 for procedure names to be exported out of the current module, 0 otherwise.

END
End of Procedure.

EOF
End of module.

Besides the export flag in PRO, six other pseudo instructions are involved with separate compilation and linking:

EXD <sym>
Export data. <sym> is exported out of this module.

IMA <sym>
Import address. IMA allows global symbol <sym> to be used before it is

defined. Note that <sym> may be defined in the same module.

IMC <sym>
Similar to IMA, but used for imported single word constants. These two different forms are necessary, because the assembler must know how much storage must be allocated if <sym> is used in CON or ROM.

FWA <sym>
Forward address. Notify the assembler that <sym> will be defined later on in this module, so that it may be used before being defined.

FWC <sym>
Similar to FWA, but for constants.

FWP <sym>
Forward procedure reference. FWP allows <sym> to be used before it is defined. <sym> must be defined in the same module and must not be exported. Normally, unknown procedure names are entered in the undefined global reference table, so that their names will be known outside this module. Procedure names introduced by FWP are treated differently, however, to prevent their being exported.

Three other pseudo instructions provide miscellaneous features:

LET <sym>,<arg>
Assembly time assignment of the second operand to the first one.

EXC <num1>,<num2>
Two blocks of instructions preceding this one are interchanged before being assembled. <num1> gives the number of lines of the first block. <num2> gives the number of lines of the second one. Blank and pure comment lines do not count.

MES <num>,<val>*
A special type of comment. Used by compilers to communicate with the optimizer, assembler, etc. as follows:

MES 0 -
An error has occurred, stop assembly.

MES 1 -
Suppress optimization

MES 2 -
Use virtual memory (EM-2)

MES 3,<num1>,<num2> -
Indicates that a local variable is never referenced indirectly. <num1> is offset in bytes from LB. <num2> indicates the class of the variable.

MES 4 -
Number of source lines (for profiler).

MES 5 -
Floating point used.

MES 6,<val>* -
Comment. Used to provide comments in compact assembly language (see below).

12. ASSEMBLY LANGUAGE INSTRUCTION LIST

For each instruction in the list the range of operand values in the assembly language is given. These ranges are all subranges of -32768..32767 and are indicated by letters:

m: full range, i.e. -32768..32767
n: 0..32767
x: 0..32766 and even
y: 1 or (2..32766 and even)
z: -32768..32766 and even
p: 2..32766 and even
r: 0, 1 or 2

The letters should not be confused with the letters used in the EM-1 instruction table in appendix 2. Instructions that check for undefined operands and underflow or overflow are indicated by (*).

GROUP 1: LOAD

LOC m - Load constant (i.e. push it onto the stack)
LNC m - Load negative constant
LOL x - Load local word x
LOE x - Load external word x
LOP x - Load word pointed to by x-th local
LAI y - Load auto increment y bytes (address of pointer on stack)
LOF m - Load offsetted. (top of stack + m yield address)
LAL x - Load address of local
LAE x - Load address of external
LEX n - Load lexical. (address of LB n static levels back)
LOI y - Load indirect y bytes (address is popped from the stack)
LOS - Load indirect (pop byte count, address; count is 1 or even)
LDL x - Load double local (two consecutive locals are stacked)
LDE x - Load double external (two consecutive externals are stacked)
LDF m - Load double offsetted (top of stack + m yield address)

GROUP 2: STORE

STL x - Store local
STE x - Store external
STP x - Store into word pointed to by x-th local
SAI y - Store auto increment y bytes (address of pointer on stack)
STF m - Store offsetted
STI y - Store indirect y bytes (pop address, then data)
STS - Store indirect (pop byte count, then address, then data)
SDL x - Store double local
SDE x - Store double external
SDF m - Store double offsetted

GROUP 3: SINGLE PRECISION INTEGER ARITHMETIC

ADD - Addition (*)
SUB - Subtraction (*)
MUL - Multiplication (*)

```
  DIV   - Division (*)
  MOD   - Modulo i.e.remainder (*)
  NEG   - Negate (two's complement) (*)
  SHL   - Shift left (*)
  SHR   - Shift right (*)

GROUP 4: DOUBLE PRECISION ARITHMETIC   (Format not defined)

  DAD   - Double add (*)
  DSB   - Double Subtract (*)
  DMU   - Double Multiply (*)
  DDV   - Double Divide (*)
  DMD   - Double Modulo (*)

GROUP 5: FLOATING POINT ARITHMETIC   (Format not defined)

  FAD   - Floating add (*)
  FSB   - Floating subtract (*)
  FMU   - Floating multiply (*)
  FDV   - Floating divide (*)
  FIF   - Floating multiply and split integer and fraction part (*)
  FEF   - Split floating number in exponent and fraction part (*)

GROUP 6: POINTER ARITHMETIC

  ADl m - Add the constant m to pointer on top of stack
  PAD   - Pointer add; pop integer, then pointer, push sum as pointer
  PSB   - Subtract two pointers (in same fragment) and push diff as integer

GROUP 7: INCREMENT/DECREMENT/ZERO

  INC   - Increment top of stack by 1 (*)
  INL x - Increment local (*)
  INE x - Increment external (*)
  DEC   - Decrement top of stack by 1 (*)
  DEL x - Decrement local (*)
  DEE x - Decrement external (*)
  ZRL x - Zero local
  ZRE x - Zero external

GROUP 8: CONVERT

  CID   - Convert integer to double (*)
  CDI   - Convert double to integer (*)
  CIF   - Convert integer to floating (*)
  CFI   - Convert floating to integer (*)
  CDF   - Convert double to floating (*)
  CFD   - Convert floating to double (*)

GROUP 9: LOGICAL

  AND p - Boolean and on two groups of p bytes
  ANS   - Boolean and; number of bytes is first popped from stack
  IOR p - Boolean inclusive or on two groups of p bytes
  IOS   - Boolean inclusive or; nr of bytes is first popped from stack
  XOR p - Boolean exclusive or on two groups of p bytes
  XOS   - Boolean exclusive or; nr of bytes is first popped from stack
  COM p - Complement (one's complement of top p bytes)
  COS   - Complement; first pop number of bytes from stack
  ROL   - Rotate left
  ROR   - Rotate right

GROUP 10: SETS

  INN p - Bit test on p byte set (bit number on top of stack)
  INS   - Bit test; first pop set size, then bit number
  SET p - Create singleton p byte set with bit n on (n is top of stack)
  SES   - Create singleton set; first pop set size, then bit number

GROUP 11: ARRAY

  LAR x - Load array element
  LAS   - Load array element;  first pop ptr to descriptor from stack
  SAR x - Store array element
  SAS   - Store array element;  first pop ptr to descriptor from stack
  AAR x - Load address of array element
  AAS   - Load address;  first pop pointer to descriptor from stack

GROUP 12: COMPARE

  CMI   - Compare 2 integers. Push negative, zero, positive for <, = or >
  CMD   - Compare 2 double integers
  CMF   - Compare 2 reals
  CMU p - Compare 2 blocks of p bytes each
  CMS   - Compare 2 blocks of bytes;  pop byte count
  CMP   - Compare 2 pointers

  TLT   - True if less, i.e. iff top of stack < 0
  TLE   - True if less or equal, i.e. iff top of stack <= 0
  TEQ   - True if equal, i.e. iff top of stack = 0
  TNE   - True if not equal, i.e. iff top of stack non zero
  TGE   - True if greater or equal, i.e. iff top of stack >= 0
  TGT   - True if greater, i.e. iff top of stack > 0

GROUP 13: BRANCH

  BRF n - Branch forward unconditionally n bytes
  BRB n - Branch backward unconditionally n bytes

  BLT n - Forward branch less (pop 2 words, branch if top > second)
  BLE n - Forward branch less or equal
  BEQ n - Forward branch equal
  BNE n - Forward branch not equal
  BGE n - Forward branch greater or equal
  BGT n - Forward branch greater

  ZLT n - Forward branch less than zero (pop 1 word, branch negative)
  ZLE n - Forward branch less or equal to zero
  ZEQ n - Forward branch equal zero
  ZNE n - Forward branch not zero
```

```
  ZGE n - Forward branch greater or equal zero
  ZGT n - Forward branch greater than zero

GROUP 14: PROCEDURE CALL

  MRK n - Mark stack (n = change in static depth of nesting - 1)
  MRS   - Mark stack; first pop the static link from the stack
  CAL n - Call procedure (with descriptor n)
  CAS   - Call indirect;  first pop procedure number from stack
  RET x - Return (function result consists of top x bytes)
  RES   - Like RET, but size of result on top of stack

GROUP 15: MISCELLANEOUS

  BEG z - Begin procedure (reserve z bytes for locals)
  BES   - Like BEG, except first pop z from stack
  BLM x - Block move x bytes; first pop destination addr, then source addr
  BLS   - Block move; like BLM, except first pop x, then addresses
  CSA   - Case jump; address of jump table at top of stack
  CSB   - Table lookup jump; address of jump table at top of stack
  DUP p - Duplicate top p bytes
  DUS   - Like DUP, except first pop p
  EXG   - Exchange top 2 words
  HLT   - Halt the machine ( Exit status on the stack )
  LIN n - Line number (external 0 := n)
  LNI   - Line number increment
  LOR r - Load register (0=LB, 1=SP, 2=HP)
  MON   - Monitor call
  NOP   - No operation
  RCK x - Range check; descriptor at (external) x; trap on error
  RCS   - Like RCK, except first pop x from stack
  RTT   - Return from trap
  SIG   - Trap errors to proc nr on top of stack (-2 resets default).  Static
          link of procedure is below procedure number.  Old values returned
  STR r - Store register (0=LB, 1=SP, 2=HP)
  TRP   - Cause trap to occur (Error number on stack)
```

13. KERNEL INSTRUCTION SET

Many of the instructions presented in the previous chapter are replacements for a small sequence of basic instructions. The basic instructions form less than half of the complete instruction set. Only a few basic instructions have operands. Most of them fetch their arguments from the stack. Very few basic instructions are provided to load and store objects.

For each of the groups of instructions the basic ones are given:

```
GROUP  1: LOC, LAE, LEX, LOS
GROUP  2: STS
GROUP  3: ADD, SUB, MUL, DIV, SHL, SHR
GROUP  4: DAD, DSB, DMU, DDV
GROUP  5: FAD, FSB, FMU, FDV, FIF, FEF
GROUP  6: PAD, PSB
GROUP  7: -
GROUP  8: CID, CDI, CDF, CFD
GROUP  9: ANS, IOS, XOS, COS, ROL, ROR
GROUP 10: INS, SES
GROUP 11: AAS
GROUP 12: CMI, CMD, CMF, CMS, CMP, TGT, TLT, TEQ
GROUP 13: DRB, ZNE
GROUP 14: MRS, CAS, RES
GROUP 15: BES, BLS, CSA, CSB, DUS, EXG, HLT, LOR, MON, NOP, RCS,
          RTT, SIG, STR, TRP
```

Almost all the other instructions can be replaced in the assembly language by a short equivalent sequence of simpler instructions. By applying these replacements recursively a sequence of basic instructions can be found.

```
GROUP 1:
  LNC m  =  LOC -m
  LOL x  =  LAL x  +  LOI 2
  LOE x  =  LAE x  +  LOI 2
  LOP x  =  LOL x  +  LOI 2
  LAI y  =  DUP 2  +  DUP 2  +  LOI 2  +  ADI y  +  EXG    +  STI 2  +  LOI y
  LOF m  =  ADI m  +  LOI 2
  LAL x  =  LEX 0  +  ADI x
  LOI y  =  LOC y  +  LOS
  LDL x  =  LAL x  +  LOI 4
  LDE x  =  LAE x  +  LOI 4
  LDF m  =  ADI m  +  LOI 4

GROUP 2:
  STL x  =  LAL x  +  STI 2
  STE x  =  LAE x  +  STI 2
  STP x  =  LOL x  +  STI 2
  SAI y  =  DUP 2  +  DUP 2  +  LOI 2  +  ADI y  +  EXG    +  STI 2  +  STI y
  STF m  =  ADI m  +  STI 2
  STI y  =  LOC y  +  STS
  SDL x  =  LAL x  +  STI 4
  SDE x  =  LAE x  +  STI 4
  SDF m  =  ADI m  +  STI 4
```

```
GROUP 3:
  MOD    =  DUP 4  +  DIV    +  MUL    +  SUB
  NEG    =  LOC 0  +  EXG    +  SUB

GROUP 4:
  DMD    =  DUP 8  +  DDV    +  DMU    +  DSB

GROUP 6:
  ADI m  =  LOC m  +  PAD

GROUP 7:
  INC    =  LOC 1  +  ADD
  INL x  =  LOL x  +  INC    +  STL x
  INE x  =  LOE x  +  INC    +  STE x
  DEC    =  LOC 1  +  SUB
  DEL x  =  LOL x  +  DEC    +  STL x
  DEE x  =  LOE x  +  DEC    +  STE x
  ZRL x  =  LOC 0  +  STL x
  ZRE x  =  LOC 0  +  STE x

GROUP 8:
  CIF    =  CID    +  CDF
  CFI    =  CFD    +  CDI

GROUP 9:
  AND p  =  LOC p  +  ANS
  IOR p  =  LOC p  +  IOS
  XOR p  =  LOC p  +  XOS
  COM p  =  LOC p  +  COS

GROUP 10:
  INN p  =  LOC p  +  INS
  SET p  =  LOC p  +  SES

GROUP 11:
  LAR x  =  LAE x  +  LAS
  SAR x  =  LAE x  +  SAS
  AAR x  =  LAE x  +  AAS

GROUP 12:
  CMU p  =  LOC p  +  CMS
  TLE    =  TGT    +  TEQ
  TGE    =  TLT    +  TEQ
  TNE    =  TEQ    +  TEQ

GROUP 13:
  BRF n  =  LOC 0  +  ZEQ n
  BLT n  =  CMI    +  ZLT n
  BLE n  =  CMI    +  ZLE n
  BEQ n  =  CMI    +  ZEQ n
  BNE n  =  CMI    +  ZNE n
  BGE n  =  CMI    +  ZGE n
  BGT n  =  CMI    +  ZGT n
  ZLT n  =  TLT    +  ZNE n
  ZLE n  =  TLE    +  ZNE n
  ZEQ n  =  TEQ    +  ZNE n
  ZGE n  =  TGE    +  ZNE n
  ZGT n  =  TGT    +  ZNE n

GROUP 14:
  MRK n  =  LOC n  +  MRS
  CAL n  =  LOC n  +  CAS
  RET p  =  LOC p  +  RES

GROUP 15:
  BEG z  =  LOC z  +  BES
  BLM p  =  LOC p  +  BLS
  DUP p  =  LOC p  +  DUS
  LIN n  =  LOC n  +  STE 0
  LNI    =  INE 0
  RCK x  =  LAE x  +  RCS
```

The replacements for LIN and LNI are only equivalent if they precede the first HOL in that assembly module. The replacements for LAI and SAI are rather artificial. These instructions are most likely preceded by a LAL or LAE instruction. Then they replace the sequence:

```
LAL x  +  LAI y  =  LOL x  +  DUP 2  +  ADI y  +  STL x  +  LOI y
LAE x  +  LAI y  =  LOE x  +  DUP 2  +  ADI y  +  STE x  +  LOI y
LAL x  +  SAI y  =  LOL x  +  DUP 2  +  ADI y  +  STL x  +  STI y
LAE x  +  SAI y  =  LOE x  +  DUP 2  +  ADI y  +  STE x  +  STI y
```

The replacements for LAS and SAS would even be longer, because the size of the object to be loaded or stored must be fetched from the descriptor. If the size y is known, then LAS and SAS can be replaced by:

```
LAS    =  AAS    +  LOI y
SAS    =  AAS    +  STI y
```

APPENDIX 1. OFFICIAL EM-1 MACHINE DEFINITION

{ This is an interpreter for EM-1. It serves as the official machine definition. This interpreter must run on a machine which supports 32 bit arithmetic.

Certain aspects of the definition are over specified. In particular:

1. The representation of an address on the stack need not be the numerical value of the memory location.

2. The state of the stack is not defined after a trap has aborted an instruction in the middle. For example, it is officially undefined whether the second operand of an ADD instruction has been popped or not if the first one is undefined (-32768).

3. The memory layout is implementation dependent. Only the most basic checks are performed whenever memory is accessed.

4. The format of the mark block is implementation dependent.

5. The format of the procedure descriptors is implementation dependent.

6. The result of the compare operators CMI etc. are -1, 0 and 1 here, but other negative and positive values will do and they need not be the same each time.

7. The shift count for SHL, SHR, ROL and ROR must be in the range 0 to 15. The effect of a count greater than 15 or less than 0 is undefined.

}

```
program em1(tables,prog,output);

label 9999;

const
  t13   =  8192;        { 2**13    }
  t14   = 16384;        { 2**14    }
  t15   = 32768;        { 2**15    }
  t15m1 = 32767;        { 2**15 -1 }
  t16   = 65536;        { 2**16    }
  t16m1 = 65535;        { 2**16 -1 }
  t31m1 = 2147483647;   { 2**31 -1 }

  maxcode = 8191;       { highest byte in code address space }
  maxdata = 8191;       { highest byte in data address space }

  { mark block format }
  statd   = 6;          { how far is static link from lb }
  dynd    = 4;          { how far is dynamic link from lb }
  reta    = 2;          { how far is the return address from lb }
  mrksize = 6;          { size of mark block in bytes }

  { procedure descriptor format }
  pdargs  = 0;          { offset for the number of argument bytes }
  pdbase  = 2;          { offset for the procedure base }
  pdsize  = 4;          { size of procedure descriptor in bytes }

  dsize   = 4;          { size of double precision integers }
  rsize   = 4;          { size of reals }
  { header words }
  NTEXT   = 1;
  NDATA   = 2;
  NPROC   = 3;
  ENTRY   = 4;
  NLINE   = 5;

  escape  = 0;          { escape to secondary opcodes }
  undef   = -32768;     { the range of integers is -32767 to +32767 }

  { error codes }
  ESTACK =  0; EHEAP   =  1; EILLINS =  2; EODDZ   =  3;
  ECASE  =  4; ESET    =  5; EARRAY  =  6; ERANGE  =  7;
  EIOVFL =  8; EDOVFL  =  9; EFOVFL  = 10; EFUNFL  = 11;
  EIDIVZ = 12; EFDIVZ  = 13; EIUND   = 14; EDUND   = 15;
  EFUND  = 16; ECFI    = 17; ECFD    = 18; ECDI    = 19;
  EFPP   = 20; ELIN    = 21; EMON    = 22; ECAL    = 23;
  ELAE   = 24; EMEMFLT = 25; EPTR    = 26; EPROC   = 27;
  EPC    = 28;
```

```
{-------------------------------------------------------------------------------}
{                              Declarations                                     }
{-------------------------------------------------------------------------------}

type
  bitval= 0..1;          { one bit }
  bitnr=  0..15;         { bits in machine words are numbered 0 to 15 }
  byte=   0..255;        { memory is an array of bytes }
  offset= 0..t15m1;      { positive integers are offsets }
  adr=    0..t16m1;      { a machine word interpreted as an address }
  word=  -t15..t15m1;    { a machine word interpreted as a signed integer }
  full=  -t16m1..t16m1; { intermediate results need this range }
  double=-t31m1..t31m1; { double precision range }
  bftype= (andf,iorf,xorf); { tells which boolean operator needed }
  iflags= (mini,short,xbit,ybit,zbit);
  ifset=  set of iflags;

  mnem = ( NON,
           AAR, AAS, ADD, ADI,XAND, ANS, BEG, BEQ, BES, BGE,
           BGT, BLE, BLM, BLS, BLT, BNE, BRB, BRF, CAL, CAS,
           CDF, CDI, CFD, CFI, CID, CIF, CMD, CMF, CMI, CMP,
           CMS, CMU, COM, COS, CSA, CSB, DAD, DDV, DEC, DEE,
           DEL,XDIV, DMD, DMU, DSB, DUP, DUS, EXG, FAD, FDV,
           FEF, FIF, FMU, FSB, HLT, INC, INE, INL, INN, INS,
           IOR, IOS, LAB, LAE, LAI, LAL, LAR, LAS, LDE, LDF,
           LDL, LEX, LIN, LNC, LNI, LOC, LOE, LOF, LOI, LOL,
           LOP, LOR, LOS, LSA,XMOD, MON, MRK, MRS, MRX, MUL,
           MXS, NEG, NOP, NUL, PAD, PSB, RCK, RCS, RES, RET,
           ROL, ROR, RTT, SAI, SAR, SAS, SDE, SDF, SDL, SES,
          XSET, SHL, SHR, SIG, STE, STF, STI, STL, STP, STR,
           STS, SUB, TEQ, TGE, TGT, TLE, TLT, TNE, TRP, XOR-
           XOS, ZEQ, ZGE, ZGT, ZLE, ZLT, ZNE, ZRE, ZRL);

  dispatch = record
               iflag: ifset;
               instr: mnem;
               implicit: word
             end;


var
  code: packed array[0..maxcode] of byte;      { code space }
  data: packed array[0..maxdata] of byte;      { data space }
  pc,lb,sp,hp,pd: adr;  { internal machine registers }
  i: integer;           { integer scratch variable }
  s,t,k: word;          { scratch variables }
  j:offset;             { scratch variable used as index }
  a,b:adr;              { scratch variable used for addresses }
  dt,ds:double;         { scratch variables for double precision }
  rt,rs,x,y:real;       { scratch variables for real }
  found:boolean;        { scratch }
  opcode: byte;         { holds the opcode during execution }
  escaped: boolean;     { true for escaped opcodes }
  cutoff: byte;         { opcode of first call in alternate context }
  dispat: array[boolean,byte] of dispatch;
  insr: mnem;           { holds the instructionnumber }
  normalmap: boolean;   { true except when in alternate context }
  halted: boolean;      { normally false. set to true by halt instruction }
  exitstatus:word;      { parameter of HLT }
  uerrorlb:adr;         { static link of error procedure }
  uerrorproc:adr;       { number of user defined error procedure }
  header: array[1..8] of adr;

  tables: text;         { description of EM-1 instructions }
  prog: file of byte;   { program and initialized data }



{-------------------------------------------------------------------------------}
{                          Various check routines                               }
{-------------------------------------------------------------------------------}

{ Only the most basic checks are performed. These routines are inherently
  implementation dependent. }

procedure trap(n:byte); forward;

procedure oddchkadr(a:adr);
begin if (a>maxdata) or ((a>sp) and (a<hp)) then trap(EPTR) end;

procedure chkadr(a:adr);
begin if odd(a) then trap(EPTR); oddchkadr(a) end;

procedure newpc(a:adr);
begin if (a<0) or (a>pd) then trap(EPC); pc:=a end;

procedure newsp(a:adr);
begin if (a<lb-2) or (a>=hp) or odd(a) then trap(ESTACK); sp:=a end;

procedure newlb(a:adr);
begin if (a>sp+2) or odd(a) then trap(ESTACK); lb:=a end;

procedure newhp(a:adr);
begin if (a<=sp) or (a>maxdata+1) or odd(a) then trap(EHEAP); hp:=a end;

function argi(w:word):word;
begin if w = undef then trap(EIUND); argi:=w end;

function argn(w:word):word;
begin if w<0 then trap(EILLINS); argn:=w end;

function argx(w:word):word;
begin if (w<0) or (w>=t15) or odd(w) then trap(EILLINS); argx:=w end;

function argp(w:word):word;
begin if odd(w) or (w<=0) or (w>=t15) then trap(EILLINS); argp:=w end;

function argy(w:word):word;
```

```
begin if w=1 then argy:=1 else argy:=argp(w) end;

function argz(w:word):word;
begin if odd(w) or (w<-t15) or (w>=t15) then trap(EILLINS); argz:=w end;

function chkovf(z:double):word;
begin if abs(z) >= t15 then trap(EIOVFL); chkovf:=z end;
```

```
{-------------------------------------------------------------------------}
{                        Memory access routines                           }
{-------------------------------------------------------------------------}

{ memw returns a machine word as a signed integer: -32768 <= memw <= +32767
  mema returns a machine word as an address : 0 <= mema <= 65535
  memb returns a single byte as a positive integer: 0 <= memb <= 255
  store(a,v) stores the word or address v at machine address a
  storeb(a,b) stores the byte b at machine address a

  memi returns a word from the instruction space: 0 <= memi <= 65535
       Note that the procedure descriptors are part of instruction space.
  nextpc returns the next byte addressed by pc, incrementing pc

  lino changes the line number word.

  All routines check to make sure the address is within range.  The word
  routines also check to see that the address is even.  If an addressing
  error is found, a trap occurs. }


function mema(a:adr):adr;
var b:adr;
begin chkadr(a); b:=data[a+1]; mema:=256*b + data[a] end;

function memw(a:adr):word;
var b:adr;
begin b:=mema(a); if b>=t15 then memw:=b-t16 else memw:=b end;

function memb(a:adr):byte;
begin oddchkadr(a); memb:=data[a] end;

procedure store(a:adr; x:full);
begin chkadr(a);
  if x < 0 then x := x+t16; { equivalent value, but positive }
  data[a] := x mod 256; data[a+1] := x div 256
end;

procedure storeb(a:adr; b:byte);
begin oddchkadr(a); data[a]:=b end;

function memi(a:adr):adr;
var b:adr;
begin
  if odd(a) or (a>maxcode) then trap(EPTR);
  b:=code[a+1]; memi:=256*b + code[a]
end;

function nextpc:byte;
begin nextpc:=code[pc]; newpc(pc+1) end;

procedure lino(w:word);
begin if (w<0) or (w>header[NLINE]) then trap(ELIN); store(0,w) end;
```

```
{-------------------------------------------------------------------------}
{                    Stack Manipulation Routines                          }
{-------------------------------------------------------------------------}

{ push puts a word or address on the stack
  popw removes a machine word from the stack and delivers it as a word
  popa removes a machine word from the stack and delivers it as an address
  pushd pushes a double precision number on the stack
  popd removes 2 machine words and returns a double precision integer
  pushr pushes a real (floating point) number onto the stack
  popr removes 2 machine words and returns a real number
  pushx puts an object of arbitrary size on the stack
  popx removes an object of arbitrary size
                                                                        }

procedure push(x:full);
begin newsp(sp+2); store(sp,x) end;

function popw:word;
begin popw:=memw(sp); newsp(sp-2) end;

function popa:adr;
begin popa:=mema(sp); newsp(sp-2) end;

procedure pushd(y:double);
begin { push double integer onto the stack } newsp(sp+dsize) end;

function popd:double;
begin { pop double integer from the stack } newsp(sp-dsize); popd:=0 end;

procedure pushr(z:real);
begin { Push a real onto the stack } newsp(sp+rsize) end;

function popr:real;
begin { pop real from the stack } newsp(sp-rsize); popr:=0.0 end;

procedure pushx(size:offset; a:adr);
var i:integer;
begin
```

```
  if size=1
     then push(memb(a))
     else if odd(size) or (size<=0)
             then trap(EODDZ)
             else for i:=1 to size div 2 do push(memw(a-2+2*i))
end;

procedure popx(size:offset; a:adr);
var i:integer;
begin
  if size=1
     then begin storeb(a,memb(sp)); newsp(sp-2) end
     else if odd(size) or (size<=0)
             then trap(EODDZ)
             else for i:=1 to size div 2 do store(a+size-2*i,popw)
end;
```

```
{-------------------------------------------------------------------------}
{              Bit manipulation routines (extract, shift, rotate)         }
{-------------------------------------------------------------------------}

procedure sleft(var w:word);  { 1 bit left shift   }
begin if abs(w) >= t14 then trap(EIOVFL) else w := 2*w end;

procedure sright(var w:word);  { 1 bit right shift with sign extension }
begin if w >= 0 then w := w div 2 else w := (w-1) div 2 end;

procedure rleft(var w:word);  { 1 bit left rotate }
begin if w >= 0
        then if w < t14 then w:= 2*w else w:= 2*w-t16
        else if w >= -t14 then w := 2*w+1 else w:= 2*w+t16+1
end;

procedure rright(var w:word);  { 1 bit right rotate }
begin if odd(w)
        then if w<0 then w:=(w-1) div 2 else w := w div 2 - t15
        else if w<0 then w:=(w+t16) div 2 else w:= w div 2
end;

function bit(b:bitnr; w:word):bitval; { return bit b of the word w }
var i:bitnr;
begin for i:= 1 to b do rright(w); bit:=ord(odd(w)) end;

function bf(ty:bftype; w1,w2:word):word;  { return boolean fcn of 2 words }
var i:bitnr; j:adr;
begin j:=0;
      for i:= 15 downto 0 do
        begin j := 2*j;
              case ty of
                andf: if bit(i,w1)+bit(i,w2) = 2 then j:=j+1;
                orf: if bit(i,w1)+bit(i,w2) > 0 then j:=j+1;
                xorf: if bit(i,w1)+bit(i,w2) = 1 then j:=j+1
              end
        end;
      if j <= t15m1 then bf:=j else bf:= j - t16
end;
```

```
{-------------------------------------------------------------------------}
{                            Array indexing                               }
{-------------------------------------------------------------------------}

function arraycalc(c:adr):adr; { subscript calculation }
var j:word; size:offset; a:adr;
begin j:= popw - memw(c);
  if (j<0) or (j>memw(c+2)) then trap(EARRAY);
  size := memw(c+4);
  if (size<0) or ((size>1) and odd(size)) then trap(EODDZ);
  a := j*size+popa;
  arraycalc:=a
end;
```

```
{-------------------------------------------------------------------------}
{                        Double and Real Arithmetic                       }
{-------------------------------------------------------------------------}

{ All routines for doubles and reals are dummy routines, since the format of
  doubles and reals is not defined in EM-1.
}

function dodad(ds,dt:double):double;
begin { add two doubles } dodad:=0 end;

function dodsb(ds,dt:double):double;
begin { subtract two doubles } dodsb:=0 end;

function dodml(ds,dt:double):double;
begin { multiply two doubles } dodml:=0 end;

function doddv(ds,dt:double):double;
begin { divide two doubles } doddv:=0 end;

function dodmd(ds,dt:double):double;
begin { modulo of two doubles } dodmd:=0 end;

function dofad(x,y:real):real;
begin { add two reals } dofad:=0.0 end;

function dofsb(x,y:real):real;
begin { subtract two reals } dofsb:=0.0 end;

function dofmu(x,y:real):real;
begin { multiply two reals } dofmu:=0.0 end;
```

```
function dofdv(x,y:real):real;
begin { divide two reals } dofdv:=0.0 end;

procedure dofif(x,y:real;var intpart,fraction:real);
begin { dismember x*y into integer and fractional parts }
  intpart:=0.0;  { integer part of x*y }
  fraction:=0.0; { fractional part of x*y }
end;

procedure dofef(x:real;var mantissa:real;var.exponent:integer);
begin { dismember x into mantissa and exponent parts }
  mantissa:=0.0;  { mantissa of x }
  exponent:=0;    { exponent of x }
end;
```

```
{------------------------------------------------------------------------}
{                                Trap                                    }
{------------------------------------------------------------------------}

procedure trap;
{ This routine is invoked for overflow, and other run time errors.
  For non-fatal errors, trap returns to the calling routine
}

begin
  if uerrorlb=0 then
    begin
      writeln('error ', n:1, ' occurred without being caught');
      goto 9999
    end;
  { Deposit all interpreter variables that need to be saved on
    the stack. This includes normalmap, all scratch variables that can
    be in use at the moment and ( not possible in this interpreter )
    the internal address of the interpreter where the error occurred.
    This will make it possible to execute an RTT instruction totally
    transparent to the user program.
    It can, for example, occur within an ADD instruction that both
    operands are undefined and that the result overflows.
    Although this will generate 3 error traps it must be possible
    to ignore them all.

    For simplicity just the normalmap flag will be stacked here }

  push(ord(normalmap));
  { Now simulate the effect of an MRS instruction }
  push(uerrorlb);               { push static link }
  push(lb);                     { push dynamic link }
  push(pc);                     { push return address }
  push(n);                      { push error number }
  { Now simulate the effect of a CAS instruction }
  newlb(sp); newpc(memi(pd+pdsize*uerrorproc+pdbase));
  if n in [ESTACK,EHEAP,EILLINS,EODDZ,ECASE,ECAL,EMEMFLT,EPTR,
           EPROC,EPC]
     then goto 9999;
end;

procedure dortt;
var s:adr;
begin
  newpc(mema(lb-reta)); s:=lb-mrksize-2; newlb(mema(lb-dynd)); newsp(s);
  { So far this was a plain ret 0 }
  normalmap := popw = 1;
end;
```

```
{-------------------------------------------------------------------------}
{                      Initialization and debugging                       }
{-------------------------------------------------------------------------}

procedure initialize;  { start the ball rolling }
{ This is not part of the official machine definition }
const tab = '   ';
var b:boolean;
    cset:set of char;
    f:ifset;
    nmini,mbase,nshort,sbase,obase,i,j,n:integer;
    c:char;

    function readword:word;
    var b1,b2:byte; a:adr;
    begin read(prog,b1,b2); a:=b2; a:=b1+256*a;
      if a>=t15 then readword:=a-t16 else readword:=a
    end;

    function readdouble:double;
    var a,b:adr;
    begin a:=readword; b:=readword;
      { construct double out of a and b } readdouble:=0
    end;

    function readreal:real;
    var b:byte; i:integer;
        s:array[1..100] of char;
    begin i:=0;
      repeat
        read(prog,b); i:=i+1; s[i]:=chr(b)
      until b=0;
      if odd(i) then read(prog,b);  { skip padding byte }
      { construct real out of character string s } readreal:=0.0
    end;

begin
  normalmap:=true;
  halted:=false;
  exitstatus:=-1;
  uerrorlb:=0;
  uerrorproc:=0;

  { initialize tables }
  for i:=0 to maxcode do code[i]:=0;
  for i:=0 to maxdata do data[i]:=0;
  for b:=false to true do
    for i:=0 to 255 do
      with dispat[b][i] do
        begin instr:=NON; iflag:=[zbit] end;

  { read instruction table file. see appendix 2 }
  reset(tables); insr:=NON;
  repeat readln(tables) until eoln(tables);     { skip until empty line }
  repeat readln(tables) until eoln(tables);     { skip until empty line }
```



```
  readln(tables);                               { skip empty line }
  repeat
    insr:=succ(insr); cset:=[]; f:=[];
    read(tables,c,c,c,c);
    while (c=' ') or (c=tab) do read(tables,c);
    repeat
      cset:=cset+[c];
      read(tables,c)
    until (c=' ') or (c=tab);
    readln(tables,nmini,mbase,nshort,sbase,obase);
    if 'x' in cset then f:=f+[xbit];
    if 'y' in cset then f:=f+[ybit];
    if 'z' in cset then
      with dispat['s' in cset][obase] do
        begin iflag:=f+[zbit]; instr:=insr end
    else
      begin
        with dispat['l' in cset][obase] do
          begin iflag:=f; instr:=insr end;
        for i:=0 to nshort-1 do
          with dispat['s' in cset][sbase+i] do
            begin iflag:=f+[short]; instr:=insr; implicit:=256*i end;
        if insr=CAL then cutoff:=mbase else
          for i:=0 to nmini-1 do
            with dispat[false][mbase+i] do
              begin iflag:=f+[mini]; instr:=insr;
                implicit:=i+ord('o' in cset)
              end;
      end;
  until eoln(tables);

  { read in program text, data and procedure descriptors }
  reset(prog);
  for i:=1 to 8 do n:=readword;  { skip first header }
  for i:=1 to 8 do header[i]:=readword;  { read second header }
  lb:=0; hp:=maxdata+1; sp:=0; lino(0);
  { read program text }
  for i:=1 to header[NTEXT] do read(prog, code[i-1]);
  { read data blocks }
  for i:=2 to readword do push(undef);  { ABS block }
  for i:=2 to header[NDATA] do
    begin n:=readword;
      if n>=0 then
        for j:=1 to n do push(undef)
      else
        begin j:=(n+t15) div t13; n:=(n+t15) mod t13;
          case j of
          0, { words }
          1: { pointers }
             for j:=1 to n do push(readword);
          2: { double integers }
             for j:=1 to n do pushd(readdouble);
          3: { reals as character strings }
             for j:=1 to n do pushr(readreal);
          end
```

```
        end
    end;
  { read descriptor table }
  pd:=header[NTEXT];
  for i:=1 to header[NPROC]*pdsize do read(prog,code[pd+i-1]);
  { call the entry point routine }
  push(maxdata);  { illegal static link }
  push(maxdata);  { illegal dynamic link }
  push(maxcode);  { illegal return address }
  newlb(sp+2);
  newpc(memi(pd + pdsize*header[ENTRY] + pdbase));
end;
```

```
{-------------------------------------------------------------------------}
{                      MAIN LOOP OF THE INTERPRETER                       }
{-------------------------------------------------------------------------}
```

{ It should be noted that the interpreter (microprogram) for an EM-1 machine can be written in two fundamentally different ways: (1) the instruction operands are fetched in the main loop, or (2) the instruction operands are fetched after the 256 way branch, by the execution routines themselves. In this interpreter, method (1) is used to simplify the description of execution routines. The dispatch table dispat is used to determine how the operand is encoded. There are 4 possibilities:

0. There is no operand
1. The operand and instruction are together in 1 byte (mini)
2. The operand is one byte long and follows the opcode byte(s)
3. The operand is two bytes long and follows the opcode byte(s)

In this interpreter, the main loop determines the operand type, fetches it, and leaves it in the global variable k for the execution routines to use. Consequently, instructions such as LOL, which use three different formats, need only be described once in the body of the interpreter.

However, for a production interpreter, or a hardware EM-1 machine, it is probably better to use method (2), i.e. to let the execution routines themselves fetch their own operands. The reason for this is that each opcode uniquely determines the operand format, so no table lookup in the dispatch table is needed. The whole table is not needed. Method (2) therefore executes much faster.

However, separate execution routines will be needed for LOL with a one byte offset, and LOL with a two byte offset. It is to avoid this additional clutter that method (1) is used here. In a production interpreter, it is envisioned that the main loop will fetch the next instruction byte, and use it as an index into a 256 word table to find the address of the interpreter routine to jump to. The routine jumped to will begin by fetching its operand, if any, without any table lookup, since it knows which format to expect. After doing the work, it returns to the main loop by jumping indirectly to a register that contains the address of the main loop. When the alternate context is entered (after the MRX or MXS instructions), this register is reloaded so that an alternate main loop is used, with an alternate branch table. A slight variation on this idea is to have the register contain the address of the branch table, rather than the address of the main loop.

Another issue is whether the execution routines for LOL 0, LOL 2, LOL 4, etc. should all have distinct execution routines. Doing so provides for the maximum speed, since the operand is implicit in the routine itself. The disadvantage is that many nearly identical execution routines will then be needed. Another way of doing it is to keep the instruction byte fetched from memory (LOL 0, LOL 2, LOL 4, etc.) in some register, and have all the LOL mini format instructions branch to a common routine. This routine can then determine the operand by subtracting the code for LOL 0 from the register, leaving the true operand in the register (as a word quantity of course). This method makes the interpreter smaller, but is a bit slower.

To make this important point a little clearer, consider how a production interpreter for the PDP-11 might appear. Let us assume the following opcodes have been assigned:

```
30: LOL 0
31: LOL 2     (2 bytes, i.e. next word)
32: LOL 4
33: LOL 6
34: LOL b     (format with a one byte offset)
35: LOL w     (format with a one word, i.e. two byte offset)
```

Further assume that each of the 6 opcodes will have its own execution routine, i.e. we are making a tradeoff in favor of fast execution and a slightly larger interpreter.

Register r5 is the em1 program counter.
Register r4 is the em1 LB register
Register r3 is the em1 SP register (the stack grows toward high core)
Register r2 contains the interpreter address of the main loop

The main loop looks like this:

```
     movb (r5)+,r0          /fetch the opcode into r0 and increment r5
     asl r0                 /shift r0 left 1 bit. Now: -256<=r0<=+254
     jmp *table(r0)         /jump to execution routine
```

Notice that no operand fetching has been done. The execution routines for the 6 sample instructions given above might be as follows:

```
lol0: mov (r4),(sp)+        /push local 0 onto stack
      jmp (r2)              /go back to main loop
lol2: mov 2(r4),(sp)+       /push local 2 onto stack
      jmp (r2)              /go back to main loop
lol4: mov 4(r4),(sp)+       /push local 4 onto stack
      jmp (r2)              /go back to main loop
lol6: mov 6(r4),(sp)+       /push local 6 onto stack
      jmp (r2)              /go back to main loop
lolb: clr r0                /prepare to fetch the 1 byte operand
      bisb (r5)+,r0         /operand is now in r0
      asl r0                /r0 is now offset from LB in bytes, not words
      add r4,r0             /r0 is now address of the needed local
      mov (r0),(sp)+        /push the local onto the stack
      jmp (r2)
lolw: clr r0                /prepare to fetch the 2 byte operand
      bisb (r5)+,r0         /fetch high order byte first !!!
      swab r0               /insert high order byte in place
      bisb (r5)+,r0         /insert low order byte in place
      asl r0                /convert offset to bytes, from words
      add r4,r0             /r0 is now address of needed local
      mov (r0),(sp)+        /stack the local
      jmp (r2)              /done
```

The important thing to notice is where and how the operand fetch occurred:
lol0, lol2, lol4, and lol6, (the mini's) have implicit operands
lolb knew it had to fetch one byte, and did so without any table lookup
lolw knew it had to fetch a word, and did so, high order byte first }

```
{-------------------------------------------------------------------------}
{                              Main Loop                                  }
{-------------------------------------------------------------------------}

begin initialize;
  repeat
    opcode := nextpc;       { fetch the first byte of the instruction }
    if normalmap or (opcode<cutoff) then
      begin escaped:=opcode=escape;
        if escaped then opcode := nextpc;
        with dispat[escaped][opcode] do
          begin insr:=instr;
            if not (zbit in iflag) then
              begin
                if mini in iflag then k:=implicit else
                if short in iflag then k:=implicit+nextpc else
                  begin k:=nextpc; if k>=128 then k:=k-256;
                       k:=256*k + nextpc
                  end;
                if xbit in iflag then k:=k*2 else
                if ybit in iflag then
                  if k=0 then k:=1 else k:=k*2
              end
          end
      end
    else
      begin insr:=CAL; k:=opcode-cutoff end;



{-------------------------------------------------------------------------}
{          Routines for the individual instructions                       }
{-------------------------------------------------------------------------}

case insr of

  NON: trap(EILLINS);

  { LOAD GROUP }
  LOC: push(k);
  LNC: push(-k);
  LOL: push(memw(lb+argx(k)));
  LOE: push(memw(argx(k)));
  LOP: push(memw(mema(lb+argx(k))));
  LAI: begin k:=argy(k); a:=popa; b:=mema(a); store(a,b+k); pushx(k,b) end;
  LOF: push(memw(popa+k));
  LAL: push(lb+argx(k));
  LAE: push(argx(k));
  LEX: begin a:=lb; for j:=1 to argn(k) do a:= mema(a-statd); push(a) end;
  LOI: pushx(argy(k),popa);
  LOS: begin k:=popa; pushx(argy(k),popa) end;
  LDL: begin k:=argx(k); push(memw(lb+k)); push(memw(lb+k+2)) end;
  LDE: begin k:=argx(k); push(memw(k)); push(memw(k+2)) end;
  LDF: begin a:=popa; push(memw(a+k)); push(memw(a+k+2)) end;
```

```
{ STORE GROUP }
STL: store(lb+argx(k),popw);
STE: store(argx(k),popw);
STP: store(mema(lb+argx(k)),popw);
SAI: begin k:=argy(k); a:=popa; b:=mema(a); store(a,b+k); popx(k,b) end;
STF: begin a:=popa; store(a+k,popw) end;
STI: popx(argy(k),popa);
STS: begin k:=popa; popx(argy(k),popa) end;
SDL: begin k:=argx(k); store(lb+k+2,popw); store(lb+k,popw) end;
SDE: begin k:=argx(k); store(k+2,popw); store(k,popw) end;
SDF: begin a:=popa; store(a+2+k,popw); store(a+k,popw) end;


{ SINGLE PRECISION ARITHMETIC }
ADD: begin t:=argi(popw); s:= argi(popw); push(chkovf(s+t)) end;
SUB: begin t:=argi(popw); s:= argi(popw); push(chkovf(s-t)) end;
MUL: begin t:=argi(popw); s:= argi(popw); push(chkovf(s*t)) end;
XDIV: begin t:= argi(popw); s:= argi(popw);
           if t=0 then trap(EIDIVZ) else push(s div t)
      end;
XMOD: begin t:= argi(popw); s:=argi(popw);
           if t=0 then trap(EIDIVZ) else push(s - (s div t)*t)
      end;
 NEG: begin t:=argi(popw); push(-t) end;
 SHL: begin t:=argi(popw); s:=argi(popw);
        for i:= 1 to t do sleft(s); push(s)
      end;
 SHR: begin t:=argi(popw); s:=argi(popw);
        for i:= 1 to t do sright(s); push(s)
      end;


{ DOUBLE PRECISION ARITHMETIC }
DAD: begin dt:=popd; ds:=popd; pushd(dodad(ds,dt)) end;
DSB: begin dt:=popd; ds:=popd; pushd(dodsb(ds,dt)) end;
DMU: begin dt:=popd; ds:=popd; pushd(dodmd(ds,dt)) end;
DDV: begin dt:=popd; ds:=popd; pushd(doddv(ds,dt)) end;
DMD: begin dt:=popd; ds:=popd; pushd(dodmd(ds,dt)) end;


{ FLOATING POINT ARITHMETIC }
FAD: begin rt:=popr; rs:=popr; pushr(dofad(rs,rt)) end;
FSB: begin rt:=popr; rs:=popr; pushr(dofsb(rs,rt)) end;
FMU: begin rt:=popr; rs:=popr; pushr(dofmu(rs,rt)) end;
FDV: begin rt:=popr; rs:=popr; pushr(dofdv(rs,rt)) end;
FIF: begin rt:=popr; rs:=popr; dofif(rt,rs,x,y); pushr(y); pushr(x) end;
FEF: begin rt:=popr; dofef(rt,x,i); pushr(x); push(i) end;


{ POINTER ARITHMETIC }
ADI: push(popa+k);
PAD: begin t:=popw; push(popa+t) end;
PSB: begin a:=popa; b:=popa; push(chkovf(b-a)) end;
```

```
{ INCREMENT/DECREMENT/ZERO }
INC: push(chkovf(argi(popw)+1));
INL: begin k:=argx(k); t:=argi(memw(lb+k)); store(lb+k,chkovf(t+1)) end;
INE: begin k:=argx(k); t:=argi(memw(k)); store(k,chkovf(t+1)) end;
DEC: push(chkovf(argi(popw)-1));
DEL: begin k:=argx(k); t:=argi(memw(lb+k)); store(lb+k,chkovf(t-1)) end;
DEE: begin k:=argx(k); t:=argi(memw(k)); store(k,chkovf(t-1)) end;
ZRL: store(lb+argx(k),0);
ZRE: store(argx(k),0);


{ CONVERT GROUP }
CID: pushd(popw);
CDI: begin dt:=popd; if abs(dt) > t15m1 then trap(ECDI) else push(dt) end;
CIF: pushr(popw);
CFI: begin rt:=popr;
       if abs(rt)>t15m1-0.5 then trap(ECFI) else push(round(rt))
     end;
CDF: begin dt:=popd; pushr(dt) end;
CFD: begin rt:=popr; if abs(rt) > t31m1-0.5 then trap(ECFD) ;
       pushd( round(rt) )
     end;


{ LOGICAL GROUP }
XAND,ANS:
     begin if insr=ANS then k:=popw; k:=argp(k);
       for j:= 1 to k div 2 do
         begin t:=popw; a:=sp-k+2; store(a,bf(andf,memw(a),t)) end;
     end;
IOR,IOS:
     begin if insr=IOS then k:=popw; k:=argp(k);
       for j:= 1 to k div 2 do
         begin t:=popw; a:=sp-k+2; store(a,bf(iorf,memw(a),t)) end;
     end;
XOR,XOS:
     begin if insr=XOS then k:=popw; k:=argp(k);
       for j:= 1 to k div 2 do
         begin t:=popw; a:=sp-k+2; store(a,bf(xorf,memw(a),t)) end;
     end;
COM,COS:
     begin if insr=COS then k:=popw; k:=argp(k);
       for j:= 1 to k div 2 do
         begin store(sp-k+2*j, bf(xorf,memw(sp-k+2*j), -1)) end
     end;
ROL: begin t:=popw; s:=popw; for i:= 1 to t do rleft(s); push(s) end;
ROR: begin t:=popw; s:=popw; for i:= 1 to t do rright(s); push(s) end;


{ SET GROUP }
INN,INS:
     begin if insr=INS then k:=popw; k:=argp(k);
       t:=popw; if t<0 then trap(ESET);
       i:= t mod 16; t:= t div 16; if 2*t>=k then trap(ESET);
       s:=memw(sp-k+2+2*t); newsp(sp-k); push(bit(i,s));
```

```
    end;
XSET,SES:
    begin if insr=SES then k:=popw; k:=argp(k);
      t:=popw; if t<0 then trap(ESET);
      i:= t mod 16; t:= t div 16; if 2*t>=k then trap(ESET)
      for j:= 1 to t do push(0);
      s:=1; for j:= 1 to i do rleft(s); push(s);
      for j := 1 to k div 2-t-1 do push(0)
    end;


{ ARRAY GROUP }
LAR,LAS:
    begin if insr=LAS then k:=popa; k:=argx(k);
      pushx(memw(k+4),arraycalc(k))
    end;
SAR,SAS:
    begin if insr=SAS then k:=popa; k:=argx(k);
      popx(memw(k+4),arraycalc(k))
    end;
AAR,AAS:
    begin if insr=AAS then k:=popa; k:=argx(k);
      push(arraycalc(k))
    end;


{ COMPARE GROUP }
CMI: begin t:=popw; s:=popw;
       if s<t then push(-1) else if s=t then push(0) else push(1)
     end;
CMP: begin a:=popa; b:=popa;
      if b<a then push(-1) else if b=a then push(0) else push(1)
     end;
CMD: begin dt:=popd; ds:=popd;
       if ds<dt then push(-1) else if ds=dt then push(0) else push(1)
     end;
CMF: begin rt:=popr; rs:=popr;
       if rs<rt then push(-1) else if rs=rt then push(0) else push(1)
     end;
CMU,CMS:
     begin if insr=CMS then k:=popw; k:=argp(k);
       t:= 0; j:= 0;
       while (j < k) and (t=0) do
         begin a:= mema(sp-j); b:=mema(sp-k-j);
           if b<a then t:= -1 else if b>a then t:= 1;
           j:=j+2
         end;
       newsp(sp-2*k); push(t);
     end;

TLT: if popw <  0 then push(1) else push(0);
TLE: if popw <= 0 then push(1) else push(0);
TEQ: if popw =  0 then push(1) else push(0);
TNE: if popw <> 0 then push(1) else push(0);
TGE: if popw >= 0 then push(1) else push(0);
TGT: if popw >  0 then push(1) else push(0);


{ BRANCH GROUP }
BRF: newpc(pc+argn(k));
BRB: newpc(pc-argn(k));

BLT: begin t:=popw; if popw <  t then newpc(pc+argn(k)) end;
BLE: begin t:=popw; if popw <= t then newpc(pc+argn(k)) end;
BEQ: begin t:=popw; if popw =  t then newpc(pc+argn(k)) end;
BNE: begin t:=popw; if popw <> t then newpc(pc+argn(k)) end;
BGE: begin t:=popw; if popw >= t then newpc(pc+argn(k)) end;
BGT: begin t:=popw; if popw >  t then newpc(pc+argn(k)) end;

ZLT: if popw <  0 then newpc(pc+argn(k));
ZLE: if popw <= 0 then newpc(pc+argn(k));
ZEQ: if popw =  0 then newpc(pc+argn(k));
ZNE: if popw <> 0 then newpc(pc+argn(k));
ZGE: if popw >= 0 then newpc(pc+argn(k));
ZGT: if popw >  0 then newpc(pc+argn(k));


{ PROCEDURE CALL GROUP }

{ There are four ways to mark the  stack.  The change in static depth can
  be given as an immediate operand or the new static link can be provided
  on the stack.  Also, the instruction may switch into alternate context,
  or not. Only two of these have mnemonics, i.e. can be used by the prog-
  rammer. These mnemonics are MRK and MRS, corresponding to the immediate
  and stacked forms respectively. The decision about using alternate con-
  text is made by the assembler.  The four cases are:
     MRK: immediate, normal context
     MRX: immediate, alternate context
     MRS: stacked, normal context
     MXS: stacked, alternate context
}

MRK,MRS,MRX,MXS:
     begin if (insr=MRS) or (insr=MXS) then k:=popw; k:=argn(k);
       a:= lb; for j:= 1 to k do a:= mema(a-statd);
       push(a); push(lb); push(0);
       normalmap:=(insr=MRK) or (insr=MRS);
     end;
CAL,CAS:
     begin if insr=CAS then k:=popw; k:=argn(k);
       a:=pd+pdsize*k; t:= memi(a+pdargs); store(sp+2-t-reta,pc);
       newpc(memi(a+pdbase)); newlb(sp+2-t); normalmap:=true;
     end;
RET,RES:
     begin if insr=RES then k:=popw; k:=argx(k);
       newpc(mema(lb-reta)); a:=sp-k; b:=lb-mrksize-2;
       newlb(mema(lb-dynd));
       for j:= 1 to k div 2 do store(b+2*j,memw(a+2*j));
       newsp(b+k);
     end;
```

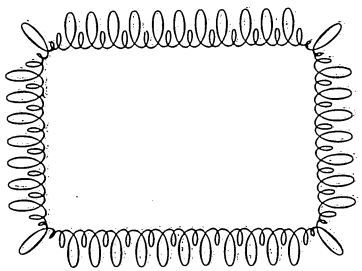
```
{ MISCELLANEOUS GROUP }
BEG,BES:
     begin if insr=BES then k:=popw; k:=argz(k);
       if k>=0
         then for j:= 1 to k div 2 do push(undef)
         else newsp(sp+k);
     end;
BLM,BLS:
     begin if insr=BLS then k:=popw; k:=argx(k);
       t:=popa; s:=popa;
       for j := 1 to k div 2 do store(t-2+2*j,memw(s-2+2*j))
     end;
CSA: begin k:=popa; b:=memi(pd+pdsize*memw(k)+pdbase);
       t:= popw - memw(k+4); s:=-1;
       if (t>=0) and (t<=memw(k+6)) then s:=memw(k+8+2*t);
       if s=-1 then s:=memw(k+2);
       if s=-1 then trap(ECASE) else newpc(b+s)
     end;
CSB: begin k:=popa; b:=memi(pd+pdsize*memw(k)+pdbase);
       t:=popw; i:=1; found:=false;
       while (i<=memw(k+4)) and not found do
         if t=memw(k+2+4*i) then found:=true else i:=i+1;
       if found then s:=memw(k+4+4*i) else s:=memw(k+2);
       if s=-1 then trap(ECASE) else newpc(b+s);
     end;
DUP,DUS:
     begin if insr=DUS then k:=popw; k:=argp(k);
       for i:=1 to k div 2 do push(memw(sp - k + 2));
     end;
EXG: begin t:=popw; s:=popw; push(t); push(s) end;
HLT: begin exitstatus:=popw; halted := true end;
LIN: lino(argn(k));
LNI: lino(memw(0)+1);
LOR: begin i:=k;
       case i of 0:push(lb); 1:push(sp); 2:push(hp) end;
     end;
MON: ; { MON will not be described here }
NOP: ;
RCK,RCS:
     begin if insr=RCS then k:=popa; k:=argx(k);
       if (memw(sp)<memw(k)) or (memw(sp)>memw(k+2)) then trap(ERANGE)
     end;
RTT: dortt;
SIG: begin a:=popa; b:=popa; push(uerrorlb); push(uerrorproc);
       uerrorproc:=a; uerrorlb:=b
     end;
STR: begin i:=k;
       case i of 0: newlb(popa); 1: newsp(popa); 2: newhp(popa) end;
     end;
TRP: trap(popw);


   end  { end of case statement }
 until halted;
9999:
  writeln('halt with exit status:',exitstatus);
end.
```

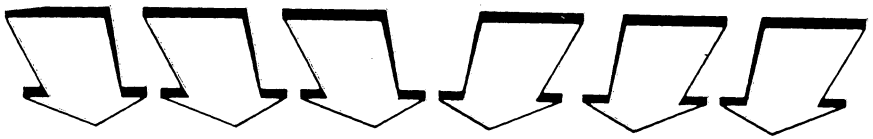
```
{*****************************************************************}
{                                                                 }
{ UNREAL ARITHMETIC -- extended precision integer arithmetic      }
{       routines for 16-bit machines.                             }
{                                                                 }
{                       Jeff Pepper                               }
{                       Three Rivers Computer Corporation         }
{                       160 N. Craig Street                       }
{                       Pittsburgh, PA 15213                      }
{                                                                 }
{                       written July 1980                         }
{                                                                 }
{ PURPOSE:                                                        }
{       This module provides routines for performing standard integer }
{       arithmetic functions with extended precision.  It is designed }
{       for use on 16-bit machines, where it effectively extends MAXINT }
{       from 32767 to roughly 256 trillion (2^48 - 1).  This is   }
{       particularly useful in financial applications, where you can }
{       store dollar amounts in tenths of a cent and still keep track }
{       of up to $256 billion.                                    }
{                                                                 }
{ IMPLEMENTATION:                                                 }
{       Numbers are of type UNREAL, a Pascal record containing 6 bytes }
{       (0..255) and a boolean indicating the sign.  The precision }
{       can be changed by changing the global constant BYTEMAX, and }
{       by changing code as noted in Uwrite.  Changing Uread is more }
{       difficult, but you probably never want to read a decimal  }
{       number larger than 15 digits anyway...                    }
{                                                                 }
{ EXCEPTIONS:                                                     }
{       The ErrorTrap procedure is called on all exceptions, which are }
{       as follows:                                               }
{               "input too long" -- too many chars in input string }
{               "input too large" -- value of input > 2^48 - 1    }
{               "no number found" -- Uread encounters a non-digit before }
{                                    finding a digit              }
{               "division by zero"                                }
{               "addition overflow"                               }
{               "mult overflow"                                   }
{       The values returned by a procedure/function are undefined if an }
{       exception is found.                                       }
{                                                                 }
{-----------------------------------------------------------------}
{          The following operations are available:                }
{                                                                 }
{ Unegate (a: unreal)                           a := -a           }
{ UUadd  (a,b: unreal; VAR c: unreal)           c := a + b        }
{ UUsub  (a,b: unreal; VAR c: unreal)           c := a - b        }
{ UUmult (a,b: unreal; VAR c: unreal)           c := a * b        }
{ UUdiv  (a,b: unreal; VAR q,rem: unreal)       q := a DIV b;     }
{                                               rem := a MOD b    }
{                                                                 }
{ UUgreater (a,b: unreal): boolean              true iff a < b    }
{ UUequal (a,b: unreal): boolean                true iff a = b    }
{ Uzero (a: unreal): boolean                    true iff a = 0    }
{                                                                 }
{ Uread (VAR f: text; VAR num: unreal)                            }
{               reads a number in decimal form, converts to type unreal }
{ Uwrite (VAR f: text; num: unreal; fieldwidth: integer)          }
{               converts from unreal to decimal form, writes to file }
{               f, using fieldwidth specified.  Writes all '*'s if }
{               fieldwidth is too small                           }
{                                                                 }
{ IUconvert (a: integer; VAR b: unreal)                           }
{       converts integer to unreal                                }
{ UIconvert (a: unreal; VAR b: integer): boolean                  }
{       converts unreal to integer.  The function returns a false value }
{       iff a > maxint.                                           }
{                                                                 }
{*****************************************************************}

CONST   bufmax = 16;                    { size of write buffer, - 1 }
        byteMax = 5;                    { size of byte array, - 1 }

TYPE    byte =  0..255;
        unreal = RECORD
                   byt: ARRAY [0..byteMax] OF byte;
                   pos: boolean;        { true if it's non-negative }
                 END;

        realArray = ARRAY [0..byteMax] OF integer;
        writeBuf = ARRAY [0..bufmax] OF integer;
        digArray = ARRAY [0..2] OF 0..9;
        string = PACKED ARRAY [0..19] OF char;

{=================================================================}

procedure UUSub (a,b: unreal; VAR c: unreal); FORWARD;

{-----------------------------------------------------------------}

procedure ErrorTrap (str: string);

BEGIN
  writeln ('*** UNREAL ARITHMETIC ERROR: ', str);
  writeln;
END;

{-----------------------------------------------------------------}

procedure Unegate (VAR a: unreal);

BEGIN
  a.pos := NOT a.pos
END;

{-----------------------------------------------------------------}

function Uzero (num: unreal): boolean;

VAR i: integer; zip: boolean;

BEGIN
  zip := TRUE;
  FOR i := 0 to byteMax DO zip := zip AND (num.byt[i] = 0); {test all bytes}
  Uzero := zip
END;

{-----------------------------------------------------------------}

function UUequal (a,b: unreal): boolean;

VAR     i: integer; eq: boolean;

BEGIN
  eq := TRUE;
  FOR i := 0 to byteMax DO eq := eq AND (a.byt[i] = b.byt[i]);
  IF a.pos <> b.pos THEN eq := FALSE;
  {just in case both are 0, but of different sign...}
  IF Uzero(a) AND Uzero(b) THEN eq := TRUE;
  UUequal := eq
END;

{-----------------------------------------------------------------}

procedure IUconvert (a: integer; VAR u: unreal);

VAR     i: integer;

BEGIN
  FOR i := 2 to byteMax DO u.byt[i] := 0;
  u.byt[1] := ABS(a) DIV 256;
  u.byt[0] := ABS(a) MOD 256;
  u.pos := (a >= 0)
END;

{-----------------------------------------------------------------}

function UIconvert (u: unreal; VAR a: integer): boolean;

{ returns TRUE iff u is in range -32767 .. +32767 }

VAR     small: boolean;
        i: integer;

BEGIN
  small := TRUE;
  FOR i := 2 to byteMax DO small := small AND (u.byt[i] = 0);
  UIconvert := small;
  a := u.byt[1] * 256 + u.byt[0];
  IF NOT u.pos THEN a := -a
```

```
END;

{-------------------------------------------------------------------------}

function UUGreater (a,b: unreal): boolean;

VAR     loc: integer;
        state: (bigger, same, smaller);

BEGIN
  IF Uzero(a) AND Uzero(b)    THEN UUGreater := FALSE
  ELSE IF a.pos AND NOT b.pos THEN UUGreater := TRUE
  ELSE IF NOT a.pos AND b.pos THEN UUGreater := FALSE
  ELSE
    BEGIN               {at this point, a and b must have same sign}
    state := same;
    loc := byteMax;
    REPEAT
      IF a.byt[loc] > b.byt[loc] THEN state := bigger
      ELSE IF a.byt[loc] < b.byt[loc] THEN state := smaller;
      loc := loc-1;
    UNTIL (state <> same) OR (loc < 0);
    IF a.pos
      THEN UUGreater := (state = bigger)        {when both are pos.}
      ELSE UUGreater := (state = smaller);      {when both are neg.}
    END;
END;

{-------------------------------------------------------------------------}

procedure Uread (VAR f: text; VAR num: unreal);

VAR     i,strLen: integer;
        tmp: realArray;
        s1: array [0..bufmax] of char;
        s: writebuf;

BEGIN
{initialize}
FOR i := 0 to bufmax DO BEGIN s[i] := 0; s1[i] := '0' END;

WHILE f^ = ' ' DO get(f);               {skip leading spaces}
num.pos := NOT (f^ = '-');              {look for minus sign}
IF f^ IN ['-','+'] THEN get(f);         {eat leading sign}

strLen := 0;
WHILE (f^ IN ['0'..'9']) AND (strLen <= bufmax) DO
  BEGIN
  read (f, s1[strLen]);                 {read into a string of digits}
  strLen := strLen + 1;
  END;
IF strLen > bufMax THEN ErrorTrap ('input too long      ')
ELSE IF strLen = 0 THEN ErrorTrap ('input not found     ')
  ELSE
    BEGIN
    {now reverse the string and convert from chars to integers}
    FOR i := 0 to strLen-1 DO s[i] := ord(s1[strLen-i-1]) - ord('0');

    {abracadabra... convert the digit array to base 256}
    tmp[0] := s[0]       + s[1] *  10 + s[2] * 100 + s[3] * 232 +
              s[4] *  16 + s[5] * 160 + s[6] *  64 + s[7] * 128;
    tmp[1] := s[3] *   3 + s[4] *  39 + s[5] * 134 + s[6] *  66 +
              s[7] * 150 + s[8] * 225 + s[9] * 202 + s[10]* 228 +
              s[11]* 232 + s[12]*  16 + s[13]* 160 + s[14]*  64;
    tmp[2] := s[5]       + s[6] *  15 + s[7] * 152 + s[8] * 245 +
              s[9] * 154 + s[10]*  11 + s[11]* 118 + s[12]* 165 +
              s[13]* 114 + s[14]* 122;
    tmp[3] := s[8] *   5 + s[9] *  59 + s[10]*  84 + s[11]*  72 +
              s[12]* 212 + s[13]*  78 + s[14]*  16;
    tmp[4] := s[10]*   2 + s[11]*  23 + s[12]* 232 + s[13]*  24 +
              s[14]* 243;
    tmp[5] := s[13]*   9 + s[14]*  90;

    FOR i := 0 to byteMax - 1 DO
      IF tmp[i] <= 255
        THEN num.byt[i] := tmp[i]
        ELSE
          BEGIN
          tmp[i+1] := tmp[i+1] + tmp[i] DIV 256;
          num.byt[i] := tmp[i] MOD 256
          END;

    {check for high byte overflow}
    IF tmp[byteMax] <= 255
      THEN num.byt[byteMax] := tmp[byteMax]
      ELSE ErrorTrap ('input too large     ');
    END;
END;
{-------------------------------------------------------------------------}

procedure Uwrite (VAR f: text; num: unreal; fieldwidth: integer);

VAR     s: writeBuf;
        i,j: integer;
        digits: digArray;
        started, goodsize: boolean;

        {-------------------------------------------------}
        procedure GetDigits (num: byte; VAR digs: digArray);
        BEGIN
        digs[2] := num DIV 100;
        digs[1] := num MOD 100 DIV 10;
        digs[0] := num MOD 10
        END;
        {-------------------------------------------------}

BEGIN
FOR i := 0 to bufmax DO s[i] := 0;

{0th byte}
  GetDigits (num.byt[0], digits);
  FOR i := 0 to 2 DO s[i] := digits[i];

{1st byte -- multiply by 256, add to s}
  GetDigits (num.byt[1], digits);
  FOR i := 0 to 2 DO
        BEGIN
        s[2+i] := s[2+i] + digits[i] * 2;
        s[1+i] := s[1+i] + digits[i] * 5;
        s[0+i] := s[0+i] + digits[i] * 6
        END;

{2nd byte -- multiply by 65536, add to s}
  GetDigits (num.byt[2], digits);
  FOR i := 0 to 2 DO
        BEGIN
        s[4+i] := s[4+i] + digits[i] * 6;
        s[3+i] := s[3+i] + digits[i] * 5;
        s[2+i] := s[2+i] + digits[i] * 5;
        s[1+i] := s[1+i] + digits[i] * 3;
        s[0+i] := s[0+i] + digits[i] * 6
        END;

{3rd byte -- multiply by 16,777,216 and add to s}
  GetDigits (num.byt[3], digits);
  FOR i := 0 to 2 DO
        BEGIN
        s[7+i] := s[7+i] + digits[i] * 1;
        s[6+i] := s[6+i] + digits[i] * 6;
        s[5+i] := s[5+i] + digits[i] * 7;
        s[4+i] := s[4+i] + digits[i] * 7;
        s[3+i] := s[3+i] + digits[i] * 7;
        s[2+i] := s[2+i] + digits[i] * 2;
        s[1+i] := s[1+i] + digits[i] * 1;
        s[0+i] := s[0+i] + digits[i] * 6
        END;

{4th byte -- multiply by 4,294,967,296 and add to s}
  IF num.byt[4] > 0 THEN
    BEGIN
    GetDigits (num.byt[4], digits);
    FOR i := 0 to 2 DO
        BEGIN
        s[9+i] := s[9+i] + digits[i] * 4;
        s[8+i] := s[8+i] + digits[i] * 2;
        s[7+i] := s[7+i] + digits[i] * 9;
        s[6+i] := s[6+i] + digits[i] * 4;
        s[5+i] := s[5+i] + digits[i] * 9;
        s[4+i] := s[4+i] + digits[i] * 6;
        s[3+i] := s[3+i] + digits[i] * 7;
        s[2+i] := s[2+i] + digits[i] * 2;
        s[1+i] := s[1+i] + digits[i] * 9;
```

```
        s[0+i] := s[0+i] + digits[i] * 6
        END;
    END;

{5th byte -- multiply by 1,099,511,627,776 (I hope) and add to s}
  IF num.byt[5] > 0 THEN
    BEGIN
    GetDigits (num.byt[5],digits);
    FOR i := 0 to 2 DO
        BEGIN
        s[12+i] := s[12+i] + digits[i] * 1;
       {s[11+i] := s[11+i] + digits[i] * 0;}
        s[10+i] := s[10+i] + digits[i] * 9;
        s[9+i]  := s[9+i]  + digits[i] * 9;
        s[8+i]  := s[8+i]  + digits[i] * 5;
        s[7+i]  := s[7+i]  + digits[i] * 1;
        s[6+i]  := s[6+i]  + digits[i] * 1;
        s[5+i]  := s[5+i]  + digits[i] * 6;
        s[4+i]  := s[4+i]  + digits[i] * 2;
        s[3+i]  := s[3+i]  + digits[i] * 7;
        s[2+i]  := s[2+i]  + digits[i] * 7;
        s[1+i]  := s[1+i]  + digits[i] * 7;
        s[0+i]  := s[0+i]  + digits[i] * 6
        END;
  END;

{*** IF YOU INCREASE THE NUMBER OF BYTES BEYOND 0..5:  repeat the process
     as above for all higher-order bytes, using a multiplier that's
     256 * the multiplier for the next lower byte ***}

{now reduce all values to range 0..9}
  FOR i := 0 to bufmax DO
    IF s[i] > 9 THEN
      BEGIN
      s[i+1] := s[i+1] + s[i] DIV 10;
      s[i] := s[i] MOD 10
      END;

{check to see if any digits will be lost}
  goodsize := TRUE;
  FOR i := fieldwidth TO bufmax DO
    goodsize := goodsize AND (s[i]= 0);

  IF NOT goodsize
    THEN FOR i := fieldwidth-1 downto 0 DO write ('*')
    ELSE
      BEGIN
      IF fieldwidth > bufmax + 1 THEN   {pad w/ spaces on right if needed}
        BEGIN
        write (' ':fieldwidth - (bufmax + 1));
        fieldwidth := bufmax + 1;
        END;

      started := FALSE;
      FOR i := fieldwidth-1 downto 0 DO
        BEGIN
        IF (s[i] = 0) AND (NOT started) AND (i > 0)
            THEN IF (NOT num.pos) AND (s[i-1] > 0)
              THEN write ('-')                  {leading minus sign}
              ELSE write (' ')                  {leading space}
          ELSE
            BEGIN
            write (s[i]:1); started := TRUE
            END;
        END;
    END;
END;

{-------------------------------------------------------------------------}

procedure UUadd (a, b: unreal; VAR c: unreal);

VAR     i: integer;
        tmp: realArray;

BEGIN
  {first, juggle the signs}
  IF a.pos AND NOT b.pos
    THEN BEGIN Unegate(b); UUSub (a,b,c) END
  ELSE IF NOT a.pos AND b.pos
    THEN BEGIN Unegate(a); UUsub (b,a,c) END
  ELSE IF NOT a.pos AND NOT b.pos
    THEN BEGIN Unegate(a); Unegate(b); UUadd(a,b,c); Unegate(c) END
  ELSE
    BEGIN       {now we know both are positive}
    FOR i := 0 to byteMax DO tmp[i] := a.byt[i] + b.byt[i];
    FOR i := 0 to byteMax - 1 DO
      IF tmp[i] <= 255
        THEN c.byt[i] := tmp[i]
        ELSE
          BEGIN
          c.byt[i] := tmp[i] - 256;
          tmp[i+1] := tmp[i+1] + 1
          END;
    IF tmp[byteMax] <= 255
      THEN c.byt[byteMax] := tmp[byteMax]
      ELSE ErrorTrap ('addition overflow   ');
    c.pos := TRUE;
    END;
END;

{-------------------------------------------------------------------------}

Procedure UUsub {a, b: unreal; VAR c: unreal};

VAR     i: integer;
        tmp: realArray;

BEGIN
  {juggle the signs}
  IF a.pos AND NOT b.pos
    THEN BEGIN Unegate(b); UUAdd(a,b,c) END
  ELSE IF NOT a.pos AND b.pos
    THEN BEGIN Unegate(a); UUadd(a,b,c); Unegate(c) END
  ELSE IF NOT a.pos AND NOT b.pos
    THEN BEGIN Unegate(a); Unegate(b); UUsub(a,b,c); Unegate(c) END

  {now make sure a>=b}
  ELSE IF UUGreater(b,a)
    THEN BEGIN UUsub(b,a,c); Unegate(c) END
  ELSE
    BEGIN
    FOR i := 0 to byteMax DO tmp[i] := a.byt[i];
    FOR i := 0 to byteMax - 1 DO
      IF tmp[i] >= b.byt[i]
        THEN c.byt[i] := tmp[i] - b.byt[i]
        ELSE
          BEGIN
          c.byt[i] := tmp[i] + 256 - b.byt[i];
          tmp[i+1] := tmp[i+1] - 1
          END;
    c.byt[byteMax] := tmp[byteMax] - b.byt[byteMax];
    c.pos := TRUE;              {it better be!}
    END;
END;

{-------------------------------------------------------------------------}

procedure UUmult (a, b: unreal; VAR c: unreal);

VAR     i, j: integer;
        tmp: realArray;

BEGIN
  FOR i := byteMax DOWNTO 0 DO
    BEGIN
    tmp[i] := 0;
    FOR j := 0 to i DO tmp[i] := tmp[i] + (a.byt[i-j] * b.byt[j]);
    END;
  FOR i := 0 to byteMax - 1 DO
    IF tmp[i] <= 255
      THEN c.byt[i] := tmp[i]
      ELSE
        BEGIN
        c.byt[i] := tmp[i] MOD 256;
        tmp[i+1] := tmp[i+1] + (tmp[i] DIV 256)
        END;
  IF tmp[byteMax] <= 255
    THEN c.byt[byteMax] := tmp[byteMax]
    ELSE ErrorTrap ('mult overflow       ');
  c.pos := (a.pos AND b.pos) OR NOT (a.pos OR b.pos);
END;
```

```
{-------------------------------------------------------------------------}

procedure UUDiv (a,b: unreal; VAR q, rem: unreal);

VAR     shiftCt, i,j: integer;
        asize, bsize: integer;

        {-------------------------------------------------}
        function TooFar (a,b: unreal): boolean;

        VAR i,j: integer; shifted: unreal;
        BEGIN
        asize := byteMax;
        WHILE (a.byt[asize] = 0) AND (asize > 0) DO asize := asize - 1;
        bsize := byteMax;
        WHILE (b.byt[bsize] = 0) AND (bsize > 0) DO bsize := bsize - 1;
        IF asize = bsize
          THEN TooFar := TRUE
          ELSE
            BEGIN
            FOR i := byteMax downto 1 do shifted.byt[i] := b.byt[i-1];
            shifted.byt[0] := 0;
            TooFar := UUGreater (shifted, a);
            END;
        END;
        {-------------------------------------------------}

BEGIN
IF Uzero(b)
  THEN ErrorTrap ('Division by zero    ')
  ELSE
    BEGIN
    {figure out quotient's & rem's signs now, then force a and b positive}
    q.pos := (a.pos AND b.pos) OR NOT (a.pos OR b.pos);
    rem.pos := a.pos;
    a.pos := TRUE;
    b.pos := TRUE;
    FOR i := 0 to byteMax DO q.byt[i] := 0;      {initialize all 0's}

    shiftCt := 0;
    WHILE NOT TooFar (a,b) DO
      BEGIN
      FOR i := byteMax DOWNTO 1 DO b.byt[i] := b.byt[i-1];      {shift left}
      b.byt[0] := 0;
      shiftCt := shiftCt + 1;
      END ;
    FOR i := shiftCt DOWNTO 0 DO
      BEGIN
      WHILE NOT UUGreater (b,a) DO
        BEGIN
        q.byt[i] := q.byt[i] + 1;
        UUsub (a,b,a);
        END;
      IF i > 0 THEN
        BEGIN
        FOR j := 0 to byteMax - 1 DO b.byt[j] := b.byt[j+1];    {shift right}
        b.byt[byteMax] := 0;
        END;
      END;
    rem.byt := a.byt;
    END;
END;

{-------------------------------------------------------------------------}

procedure Main;

VAR a,i,f: integer;
    x,y,z,rem: unreal;
    ch: char;
    dummy: boolean;

BEGIN
REPEAT
  write ('Enter problem in form n-op-n: ');
  Uread (input, x);
  read (ch);
  Uread (input, y);
  CASE ch OF
        '>': IF UUgreater(x,y) THEN write ('greater') ELSE write ('not grtr');
        '=': IF UUequal(x,y) THEN write ('equal') ELSE write ('not equal');
        'c': BEGIN dummy := UIconvert(x,a); if dummy THEN write ('conv OK');
             write (a:10); IUconvert(a,z) END;
        '+': UUadd (x,y,z);
        '-': UUsub (x,y,z);
        '*': UUmult (x,y,z);
        '/': UUdiv (x,y,z,rem);
        END; {case}
  write ('----------------------> ');
IF ch IN ['+','-','*','/','c'] THEN Uwrite (output,z,15);
  IF ch='/' THEN BEGIN write (', rem = '); Uwrite(output,rem,10) END;
  writeln;
UNTIL false;
END;


{=========================================================================}

BEGIN
  Main
END.
```





Articles

AN EXTENSION TO PASCAL READ AND WRITE PROCEDURES

David A. Rowland
Real-Time Software Associates
2717 Hillegass Ave.
Berkeley, Calif. 94705
(415) 548-8095

Pascal READ and WRITE have several distinct actions. They convert between internal forms of data and their representations as character strings, and they direct the character strings through files. They are also the only procedures in Pascal that allow an arbitrary number of parameters of varying types.

Sometimes it is useful to have the properties of READ and WRITE separate from the file structure. For example, one may wish to convert an integer to a character string and store the string in an array. Or one may wish to take input from a keyboard directly through its input buffer address rather than defining a system handler for it.

Files in READ and WRITE are specified by being named first in the parameter list. If no file name appears, an appropriate system file is implied. The extension is to allow the first parameter in the list to be the name of a user-defined procedure. For READ it must be a procedure having a parameter list like (VAR CH:CHAR). For WRITE it must have a parameter list like (CH:CHAR).

The actions are then: for READ, every time a character is sought, the user procedure is called. It returns the character in CH. For WRITE, the user procedure is called with the character provided as the parameter.

This extension is very much in the spirit of Pascal, which elsewhere allows procedures to be passed as parameters. It may seem a slight convenience in standard Pascal, but it is an enormous aid in the multi-tasking version of Pascal which we have created. It allows one the full flexibility and familiarity of READ and WRITE in the absence of any operating system. It might be considered for other real-time and process control languages.

In this example characters derived from the variable I by WRITE are sent to the procedure CONVERT, which stores them in an array.

```
VAR
 CHARS:ARRAY(.1..10.) OF CHAR;
 C, I:INTEGER;

PROCEDURE CONVERT(CH:CHAR);
BEGIN
 IF C <= CMAX
 THEN
  BEGIN
   CHARS(.C.):=CH;
   C:=C+1;
  END;
END;

BEGIN
 C:=1;  I:=437;
 WRITE(CONVERT, I);
END.
```

The second example shows how READ can read integers directly from a hardware input buffer.

```
VAR
 I, J:INTEGER;

PROCEDURE GETCH(VAR CH:CHAR);
VAR
 RCSR ORIGIN 177560B:INTEGER;
 RBUF ORIGIN 177562B:CHAR;
BEGIN
 /*Until a char is ready, wait here*/
 WHILE RCSR = 0 DO /*nothing*/ ;
 CH:=RBUF;
END;

BEGIN
 READ(GETCH, I, J);
END.
```

PDP-11 PASCAL: THE SWEDISH COMPILER

VS

OMSI PASCAL-1

Margaret A. Kulos
Naval Underwater Systems Center
New London, Connecticut

ABSTRACT

This paper presents a comparison of Seved Torstendahl's Swedish Pascal compiler and the Oregon Minicomputer Software Inc. (OMSI) Pascal-1 compiler.

A comparison of the results of applying the Pascal Validation Suite against both compilers is reported. A discussion of the factors that need consideration in transporting programs written for one of the compilers to the other, based on the results of the validation suite, is presented.

INTRODUCTION

This paper presents a comparison of two Pascal compilers implemented on a PDP-11/70 running the RSX-11M-PLUS operating system.

A comparison of the results of applying the Pascal Validation Suite against Seved Torstendahl's Swedish Compiler and the Oregon Minicomputer Software Inc. (OMSI) Pascal compiler is reported. Both compilers are discussed in relation to the requirements of the draft Pascal standard. Specific areas where programs written for one compiler may not be compatible with the other compiler are highlighted. This paper does not discuss the differences in the I/O handling by the two compilers except for presenting the validation suite results for tests that examine I/O as stated in the draft standard.

PASCAL STANDARDIZATION

The formal effort to produce a standard for the Pascal programming language began in 1977 when a working group was formed within the British Standards Institution (BSI). In October 1978, Pascal was listed as a International Standards Organization (ISO) work item and a working draft was circulated as the ISO document (1).

The current version of the standard (the 5th working draft) is being circulated to ISO member bodies for comment. In the United States, the cognizant body is the joint ANSI X3J9-IEEE Pascal Standards Committee (2).

THE PASCAL PROCESSOR VALIDATION SUITE

The Pascal processor validation suite by A.H.J. Sale and R.A. Freak is a series of test programs written in Pascal that are designed to support the draft standard (3,4). This suite of programs may be used to validate a compiler by presenting it with a series of programs which it should or should not accept. The suite also contains a number of tests that explore implementation defined features and the quality of the processor. Processors that "pass" all the tests are likely to be well designed and relatively trouble free, although they may not be error free.

Use of the validation suite provides an opportunity to measure the quality of a processor and aids implementators in providing a correct implementation of "standard" Pascal in an effort to improve the portablity of Pascal programs.

The six classes of tests in the validation suite are conformance, deviance, implementation defined, error handling, quality, and extension.

Conformance programs are correct standard Pascal programs that should compile and execute.

Programs in the deviance class are Pascal programs that differ in subtle ways from the standard. These detect processors that:

(a) handle an extension of Pascal
(b) fail to check or limit some Pascal feature appropriately, or

(c) incorporate some common error.

Implementation defined programs detail features of the processor that are implementation dependent.

The programs in the error handling category test situations where an error should be detected. This enables documentation of undetected error conditions.

Programs that explore the quality of an implementation are classified as quality tests.

The final category of tests investigates the syntax of extensions to the language according to the conventions cited in the standard.

All test programs are labeled with a test number corresponding to the section in the standard which gives rise to the test followed by a dash and a serial number that uniquely identifies each test written for that section. For example, the test numbered 6.10-3 is the third test in the validation suite corresponding to that section of the standard numbered 6.10.

SWEDISH COMPILER VALIDATION REPORT

The following is a report of results obtained by running the Pascal Validation Suite against the Swedish Compiler Version 6. The details of the test results state the actions demonstrated by the compiler for a particular test rather than the requirements listed in the standard. Examples of syntax constructs that will cause a test to fail are provided in the descriptions only for those tests that are not self-explanatory.

Pascal Processor Identification

Computer: DEC PDP-11/70 running RSX-11M-PLUS V1 BL6

Processor: Swedish Pascal Compiler Version 6.01

Test Conditions

Tester: M.A. Kulos

Date: September 1980

Validation Suite Version: 2.2

Conformance Tests

Number of tests passed: 118
Number of tests failed: 17

Details of failed tests:

6.1.8-1 Comment is not considered to be a token separator.

PROCEDURE(*comment*)ABC; is not a legal procedure heading.

6.2.2-3 Type identifier which specifies the domain of a pointer type is not permitted to have its defining occurrence anywhere in the type definition part in which the pointer type occurs.

```
PROGRAM Name;
TYPE
node=real;
.
.
.
PROCEDURE X;
TYPE
  p=^node;
```

6.4.3.3-1 Empty field-list in variant part of record type definition is not allowed.

```
e = RECORD
  CASE married OF
  true: (spousename:string);
  false: ()
  END;
```

6.4.3.5-1 File of pointer to integer is not allowed.

```
TYPE
  i=integer;
VAR
  ptr:^i;
  filex:file of ptr;
```

6.4.3.5-3 The end of line marker is not inserted at the end of a line, if not explicitly done in a program.

6.6.3.1-5, 6.6.3.4-1 and 6.6.3.5-1 Procedure declaration is not permitted as argument to a procedure. Procedures and functions may not be passed to other procedures and functions as parameters.

```
PROCEDURE Conforms(PROCEDURE abc(x:integer));
```

Note: Version 4 of the Swedish compiler would process this statement correctly if procedure abc did not have an argument--which goes along with the Jensen and Wirth definition of a parameter list (5).

6.6.3.4-2 The environment of procedure parameters does not conform to the requirements stated in the standard. (This test did not compile because of the use of a procedure as an argument to a procedure.)

6.6.5.2-3 "TRUE" is not assigned to "EOF" if the file is empty when reset.

6.6.5.4-1 UNPACK is not implemented by the compiler.

6.6.6.2-3 The arithmetic function ARCTAN is not implemented.

6.6.6.3-1 Transfer functions TRUNC and ROUND give error... floating point number too large. (This error is due to the failure of the function DIV on a negative number rather than the implementation of the functions.)

6.8.2.4-1 Non-local GOTO statements are not allowed.

6.8.3.9-7 The use of extreme values in a FOR loop causes wraparound (overflow), - leading to an infinite loop.

```
FOR i:= MAXINT-10 to MAXINT DO something;
```

6.9.2-2 Read of a character variable is not equivalent to correctly positioning the buffer variable.

6.9.4-4 Real numbers are not correctly written to text files due to the fact that when a real number does not fit the format specified, or the fraction length is not specified, the number is written to the text file in scientific notation.

Deviance Tests

Number of deviations correctly detected: 63
Number of tests showing true extensions: 1
Number of tests not detecting erroneous deviations: 30

Details of extensions:

6.1.5-6 Lower case "e" may be used in real numbers (e.g. 1.602e-20).

Details of deviations not detected:

6.1.2-1 NIL is not implemented as a reserved word and may be redefined.

6.1.7-5 and 6.9.4-12 Packed is ignored so that packed array of char is identical to array of char.

6.1.7-6 and 6.1.7-7 Strings are compatible with bounds other than 1..n, allowing deviant programs to execute.

```
TYPE
  alpha = 'A'..'Z';
VAR
  a1 : array[1..4] of char;
  a2 : array[0..3] of char;
  a3 : array[2..5] of char;
  a4 : array[1..4] of alpha;
BEGIN
  a1:='ABCD';
(* the next three are not valid assignments*)
  a2:='EFGH';
  a3:='IJKL';
  a4:='MNOP';
```

6.1.7-8 Compatibility of subranges of char and packed arrays of char is not checked and the assignment of erroneous values is allowed.

6.10-3 The default file output is not implicitly declared and it can be redefined.

6.2.2-4 Incorrect scope allows programs that are incorrect to compile.

```
(*  'red' is used in a local procedure
before its declaration.  *)

PROGRAM Xxx;
CONST
  red=1;
PROCEDURE Yyy;
CONST
  m=red
TYPE
  colour:(yellow,green,red);
```

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

6.3-5 Signed constants are permitted in contexts other than CONST declarations.

```
Writeln(+TEN);
```

6.3-6 Scope error...constant may be used in its own declaration.

```
PROGRAM Mainprogram;
CONST
  ten=10;
PROCEDURE Localprocedure;
CONST
  ten=ten;
```

6.4.1-3 Attempt to use types in their own definition when the type with the same identifier is available in an outer scope is not detected by the compiler.

6.4.2.4-2 Real constants are permitted in a subrange declaration. (Should be limited to subrange of another ordinal type.)

6.4.3.2-2 Index type should be limited to ordinal-types. Compiler allows real bounds.

```
testarray = array [1.5..10.1] of real;
```

6.4.3.2-5 Strings are not required to have subrange of integers as an index type.

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

```
PROGRAM.........
TYPE
  colour = (red,pink,orange,yellow,
                green,blue);
  subone = red..yellow;
  subtwo = pink..blue;
VAR
  colour1 : subone;
  colour2 : subtwo;
PROCEDURE test(VAR coll:subone);
  .
  .
  .
END (*procedure*)
BEGIN   (*main program*)
  colour2:=pink;
  test(colour2)
END.

(* Colour1 and colour 2 are compatible but
   not identical.  The call to procedure
   test should fail in this example.    *)
```

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

6.4.5-4 Non-identical record types allowed as var parameters.

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

6.6.2-5 Function declaration with no assignment to function identifier is permitted.

6.7.2.2-9 Unary operaonr plus is allowed to other than numeric operands.

(e.g.)
```
CONST
  dot = '.';
    .
    .
    .
BEGIN
  WRITELN(+dot);
```

6.8.2.4-2 Jumps between branches of an IF statement are allowed.

6.8.2.4-3 Jumps between branches of a CASE statement are allowed.

6.8.3.9-2, 6.8.3.9-3, and 6.8.3.9-4 Assignment to a FOR statement control variable within the FOR loop is not detected by compiler.

6.8.3.9-9 Non-local variable at an intermediate level can be used as a FOR statement control variable.

6.8.3.9-14 Global variable (at the program level) can be used as a control variable in a FOR statement.

6.8.3.9-19 Nested FOR statements using the same control variable are not detected.

6.9.4-9 Attempt to output integers whose field width parameters are zero or negative are not detected by compiler.

Error Handling Tests

Number of errors correctly detected: 35
Number of errors not detected: 31

Details of Errors Not Detected

6.2.1-7 Local variables are not undefined at beginning of statement part.

6.4.3.3-5, 6.4.3.3-6, 6.4.3.3-7, 6.4.3.3-8 Variant un-definition is not detected, there is no checking on the tag field of variant records.

6.4.6-4 Value of expression out of closed interval of destination in assignment statement is an error and is detected at run time with a PASRUN error 12 (subscripting error) occurring. The program, however, continues to execute.

```
VAR
  Answer : array[1..5] of integer;
  i : integer;
        .
        .
        .
  i:=5;
  answer:=2*i;
```

6.4.6-6 Array subscript compatibility is not checked.

6.4.6-7 Members of a set expression not in the closed interval specified by base type of assignment destination are not detected as errors.

6.4.6-8 Assignment compatibility for sets passed as parameters is not checked.

6.5.4-1, 6.5.4-2 Pointer variable with undefined value or value NIL when de-referenced is not detected.

6.6.2-6 Undefined function result is not detected.

6.6.5.2-1 Put operation on file when EOF is false is not detected. This may occur when a file is reset (opened for read only) and written to.

6.6.5.2-6, 6.6.5.2-7 Changing current file position while buffer variable is an actual parameter to a procedure or an element of a record variable list does not produce an error message.

6.6.5.3-4, 6.6.5.3-5, 6.6.5.3-6 Dispose procedure is not implemented.

6.6.5.3-7 Variables from NEW used as operand in assignment statement or actual parameter pass undetected.

6.6.6.2-4, 6.6.6.2-5 Negative arguments passed to LN or SQRT are not detected.

6.7.2.2-3 When the second operand of DIV is zero, no error is detected.

6.7.2.2-6, 6.7.2.2-7 Result of binary integer operations not in range 0..MAXINT and 0..-MAXINT are not flagged as errors.

6.7.2.2-8 MOD zero is not detected as an error.

6.8.3.5-5 CASE statement that does not contain a constant of selected value produces no warning.

6.8.3.9-5, 6.8.3.9-6 The use of a FOR statement control variable after FOR statement without an intervening assignment or, the use of a control variable after a loop which is not entered is an error that is not detected.

6.8.3.9-17 Nested FOR statements using same control variable are not detected as errors.

6.9.2-4, 6.9.2-5 Reading integers and reals from file of text when the text is not a valid integer or real number does not produce a diagnostic. For example, the text string read as a real 'ABC123.456' is not detected as an error.

Implementation Defined Tests

The implementation defined tests in the validation suite demonstrated the following characteristics of the Swedish compiler:

- A rewrite is permitted on the output file.
- Alternate comment delimiters are implemented.
- Equivalent symbols for ^, :, and := are not allowed.
- Equivalent symbol for [ ] is implemented (i.e., (. .) is allowed).
- Alternate symbols for <, >, <=, >=, and <> are not available.
- The value of MAXINT is 32767.
- Ordinal numbers of set elements must lie in the range 0..63 or ' '..'_' for characters.
- A measure of time and space requirements of a program which is an implementation of Warshall's algorithm yields:

  space = 370 bytes (2960 bits)
  time = 1.066 seconds
  (This is in comparison to 0.81646 seconds and 143 bytes--6864 bits on a Burroughs B6700 running the B6700 Pascal compiler version 2.9.001.)
- The characteristics of the floating-point arithmetic system are determined to be:

  24 bit mantissa.
  Rounds on arithmetic.
  EPS (smallest positive number such that 1.0+EPS <> 1.0)is:
  6.4604644E-08.
  The smallest positive floating point number is: 2.9387357E-39.
  The largest positive floating point number is: 1.7014119E+38.
- The value of expressions are fully evaluated before the boolean value is determined.
- Index is selected before an expression is evaluated.
- Expression is evaluated before a pointer is de-referenced.
- The output buffer is flushed at the end of program execution.
- Real numbers are written with two exponent digits.
- Default field width values are:

Integer	8 characters
Boolean	6 characters

Real 15 characters.

A total of 18 implementation defined tests were run.

Quality Tests

Twelve quality tests were executed, producing the following observations:

- There are 10 significant characters in an identifier.
- The compiler does not assist in detecting unclosed comments.
- More than 50 types are allowed.
- More than 50 labels permitted.
- More than 100 variable declarations allowed.
- Functions SQRT, EXP, SIN, COS, LN are implemented consistently.
- Function ARCTAN is not implemented.
- Operator DIV does not handle negative values correctly.
- Warnings are not generated for impossible cases in a CASE statement.
- FOR statements may be nested at least 15 levels deep.
- FOR statement control variable may be accessed upon exit from loop (value is last value in loop).
- Recursive I/O is allowed using the same file.
- Large populated CASE statement (containing 255 constants) is allowed.

Extensions

Number of tests run = 1

The only extension test run demonstrated that the OTHERWISE clause in a CASE statement has not been implemented but has instead been modified to use the word OTHERS as a case constant.

OMSI VALIDATION REPORT

The OMSI Pascal-1 compiler was tested against the Pascal Validation Suite by Barry Smith, a member of the Oregon Software implementation/maintenance team in September 1979 (6).

Conformance Tests

Of the 137 conformance tests attempted, 15 failed. The major reasons were:

- Comment delimeters not required for pairwise matching.
- Pointer scope not handled correctly.
- Assignment to function identifier within nested module generates faulty code.
- Empty record types and cases are not allowed.
- Equal, compatible sets of different base types do not compare.
- Set of char is implemented as a 64 element set.
- Procedural parameters do not conform to draft standard proposal.
- End of file on empty temporary file not checked.
- Pack and unpack not implemented.
- Empty field specifications not allowed in record declarations.
- Conversions on reading real numbers not identical to the conversions performed by the compiler.
- Writing boolean values is incorrectly right-justified.

Deviance Tests

Forty-one of the 95 deviance tests attempted in the compiler test proved to be deviations to the standard. The basic causes were:

- Real number constants without digits after point allowed.
- Packed array of char identical to array of char
- Requirements to be a string-type are not checked.
- Empty string allowed.
- Incorrect scope allows incorrect programs to compile and execute.
- Invalid programs where function identifier is inaccessible.

- Function identifier may be assigned outside of its block.
- Packed scalars, subranges and type-identifiers are allowed.
- Non-integer subrange index types are allowed for string types.
- The use of a set of real is not detected.
- Compatible but not identical var parameters are allowed.
- Non-identical array types and pointer types allowed as var parameters.
- File assignment and records containing file components compiled as descriptor copy.
- Functions without assignment to function identifier allowed.
- GOTO statements that transfer into structured statement components are allowed.
- Control variable in a FOR statement may be from any level of the program and may be assigned a value within the statement. The same variable may also be used in nested loops.
- Use of external file (other than program parameters) 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 may be omitted.

Error tests

Of the forty-eight tests attempted, 11 detected errors while 35 of the remaining tests compiled and executed without detecting the areas where the code deviates from the standard. The basic causes of undetected errors were:

- Use of un-defined values.
- Variant undefinition.
- Assignment compatibility (except index type in arrays).
- NIL or undefined pointer de-referencing.
- Undefined function result.
- File buffer aliasing and use of file.
- Some disposing conditions with undefined values or var parameters.
- Dynamic variant record used in expression or assignment.
- Succ or pred of limiting value in type.
- Chr of very large integer.
- Overflow of integer type.
- Assignment compatibility with overlapping sets.
- Case expression with no matching label.
- Use of for statement control variable after loop termination.
- Nested loops using same control variable.

Implementation Defined Tests

The execution of the implementation defined tests showed the following results:

- The value of MAXINT is 32767.
- The set of char is not implemented (but is equivalent to the set of characters from underscore character to the back-arrow character.
- Set limits are 0 to 63.
- Standard functions are not allowed as functional parameters.
- Real representation is as follows:
  24 bit mantissa.
  Rounds on arithmetic.
  EPS = 5.96E08.
  Minimum floating point number is:
  2.393E-39.
  Maximum floating point number is:
  1.70E+38.
- Boolean expressions are evaluated fully.
- Index to array selected before expression evaluated (e.g. a[i]:=exp).
- Evaluation before dereferencing in the statement p^:=exp.
- Real numbers are written with two exponent digits.
- Default field widths are:
  Integer 7
  Boolean 5
  Real 13
- A rewrite is permitted on the output file.
- Alternate symbols are allowed only for comment delimiters.

Quality tests

Twenty-seven quality tests were attempted, with three tests failing for the following reasons:

- Could not handle program with 50 labels (infinite loop).
- The use of a real expression in the SIN/COS test generated error for lack of register.
- Fatal error when compiling 11 nested for loops.

The quality measurements resulting from the other 21 tests demonstrate the following:

- Identifiers of any length are allowed, disallowing all mis-spellings.
- Unclosed comments take the remainder as comment with no warnings.
- More than 50 types are allowed.
- Array[integer] is detected but diagnostic message produced is not a applicable warning.
- Record fields are allocated representation space in declaration order.
- More than 100 variable declarations are allowed.
- Less than 10 nested procedures are allowed.
- Mod is inconsistent for negative operands.
- No warnings generated for impossible CASE clauses.
- More than 256 case constants are allowed.
- Undefined (out-of-range) values of case expressions are possible but do not cause damage.
- No more than 3 nested WITH statements permitted.
- Textfile without EOLN at end is still printed.
- Recursive I/O allowed on same file.

COMPARISON OF VALIDATION TEST RESULTS

A comparison of the results of applying the Pascal Validation Suite to both the Swedish compiler and the OMSI Compiler produced the results shown in table 1.

CLASS	SWEDISH COMPILER	OMSI COMPILER
CONFORMANCE	87%	89%
DEVIANCE	68%	56%
ERRORHANDLING	76%	76%

Table 1

Percent of Test Results
Consistant with Draft Standard

The results show that both compilers conform relatively well to the standard definition in accepting "correct" programs. They are also comparable in error detection.

The OMSI compiler appears to deviate in more cases than the Swedish compiler in that it accepts more syntax constructs that are not allowable according to the definitions.

The following is a list of the areas where the two compilers differed in the conformance and deviance tests of the Pascal Validation Suite. The details for each instance are available in the validation reports for these compilers. It is important to note that these factors need consideration when trying to ensure that programs written for one compiler may be transported to the other.

- The Swedish compiler allows redefinition of NIL.
- The OMSI compiler allows a decimal point not followed by a digit.

- Comments are not allowed as token separators in the Swedish compiler.
- The Swedish compiler permits lower case "e" to be used in real numbers.
- The OMSI compiler comment delimiters do not have to be a pairwise match.
- The OMSI compiler allows invalid programs with inaccessible function identifiers and functions that attempt assignments outside their blocks. Assignment to a function identifier from within a nested procedure or function generates bad code.
- The OMSI compiler allows signed characters, strings, scalars, and enumerated types.
- The Swedish compiler permits a constant to be used in its own declaration.
- Real constants are allowed in subrange declarations by the Swedish compiler.
- The OMSI compiler allows packed scalars, subranges (i.e., not restricted to structures), and packed type identifiers.
- The Swedish compiler allows real bounds as an index type.
- The Swedish compiler allows the use of undefined variants in a record.
- The OMSI compiler does not detect the use of a set of reals as erroneous.
- A file of pointer to integer is not allowed by the Swedish compiler.
- The Swedish compiler allows non-identical record types as var parameters.
- Compatability of file types and records containing file components is allowed by the OMSI compiler.
- Equal compatible sets of different base types do not compare as equal in the OMSI compiler.
- Unpack is not supported by the Swedish compiler.
- The Swedish compiler does not support the ARCTAN function.
- Non-local GOTO statements are not allowed by the Swedish compiler.
- In the Swedish compiler, the assignment does not follow the expression evaluation in a FOR statement.
- The control variable in a FOR statement is allowed as a formal parameter by the OMSI compiler.
- Reading a character variable is not equivalent to correctly positioning the buffer variable in the Swedish compiler.
- The Swedish compiler does not allow redefining the default file at a local level.
- Real numbers are not correctly written to text files by the Swedish compiler because the format defaults to scientific notation when the real number does not fit the format specified.
- Negative field widths give undesired output and issue no warning in the Swedish compiler. The OMSI compiler uses the absolute value of the width and gives an octal interpretation of the number.
- The OMSI compiler ignores program parameters, allowing the use of an external file not declared.
- The entire program heading may be omitted and not detected by the OMSI compiler.

The Swedish compiler and the OMSI compiler generated similar results in the validation suite tests for standard implementation defined features and quality. The following is a list of areas where the two compilers differed. The reader is again referenced to the validation suite reports for the details of the test results for each compiler.

- The Swedish compiler allows (. .) as a substitute for [ ].
- The OMSI compiler default output field width for integers is 7 characters, whereas the Swedish compiler default is 8.
- The OMSI compiler default output field width for boolean values is 5 characters, whereas the Swedish compiler default is 6.
- The OMSI compiler default output field width for reals is 13 characters, whereas the Swedish compiler default is 15.
- Identifiers are significant to 10 characters in the Swedish compiler. The OMSI compiler has no limit.
- The OMSI compiler MOD function is inconsistently implemented for negative numbers.
- The Swedish compiler DIV function is inconsistently implemented for negative numbers.

ADDITIONAL NOTES

In further examination of the results of the tests of the validation suite for the OMSI and Swedish Compilers, it is important to note that there are areas in which both compilers disagree with the proposals of the draft standard. These items should also be considered when writing programs for either compiler in order to attain code that is reasonably compiler independent. The following is a list of features found

in both compilers that do not agree with the draft standard.

- Empty strings are allowed.
- Packed is ignored. A packed array of char is identical to an array of char and similarly with other structures.
- String type requirements are not checked.
- I/O files can be redefined (i.e., not implicitly declared at the program level.
- Pointer scope is not handled correctly.
- A function identifier may be assigned a value outside of its block.
- The unary operator "+" is allowed with a constant identifier.
- String types are allowed to have non-integer subrange index types.
- Empty record types with semicolons and empty case variants are not permitted.
- Var parameters that are compatible but not identical are allowed.
- Non-identical array types and non-identical pointer types are allowed as var parameters.
- A function definition with no assignment to the function identifier is allowed.
- Only the procedure parameters as defined by Jensen and Wirth are allowed.
- End-of-file is not checked on an empty temporary file.
- GOTO statements are allowed to transfer into structured statement components.
- Assignment to a FOR control variable is allowed within the FOR statement.
- The FOR statement control variable is allowed to be program global.
- Nested loops using the same control variable produces an infinite loop.
- The Swedish compiler allows an otherwise clause in a case statement, using the word OTHERS as a case constant (the standard proposes OTHERWISE). The OMSI compiler, however, allows an ELSE clause similar to the ELSE clause of an IF statement, rather than a case label.

CONCLUSION

This paper has no conclusion. The statistical differences comparing both compilers to the draft standard are not absolute measures of the "correctness" of a compiler and should not be viewed as such. The intent of this discussion has been to present the differences between the Swedish Pascal Compiler and the OMSI Pascal-1 Compiler from a user perspective, considering what syntax construct are particular to a certain compiler and should not be used in programs that are intended to be transportable. It would be difficult to say that one compiler is better than the other based solely on the information presented in this paper.

REFERENCES

(1) Addyman, A.M., "Pascal Standardisation", Pascal News, No. 18, 1980.

(2) Addyman A.M., "A Draft Proposal for Pascal", Pascal News, No. 18, 1980.

(3) Winchmann, B.A., and Sale, A.H.J., "A Pascal Processor Validation Suite", (document accompanying Pascal Validation Suite).

(4) Sale, A.H.J., "The Pascal Validation Suite - Aims and Methods", Pascal News, No. 16, 1980.

(5) Jensen, Kathleen and Wirth, Niklaus, Pascal User Manual and Report, Springer-Verlag, New York, 1974.

(6) "Three Sample Validation Reports", Pascal News, No. 16, 1980.

ACKNOWLEDGEMENTS

The author wishes to acknowledge the assistance of E. Wade Scannell of Shearwater, Inc., in analyzing the results of the validation suite applied to the Swedish Pascal compiler.


```
 TEXT VERSION:2.50-01  INSTALLED  AUG  1980
 ON:  MEAP 11/70 SYSTEM
  FOR HELP CALL:
 STEPHEN P. PACHECO (4730) OR ROY E. TOZIER (4754)
        START OF RUN: 16:03:38     26-OCT-80
        END OF RUN 16:04:13 ELAPSED WALL TIME=     30.93 SECONDS.


 COMMAND LINE SUPPLIED TO TEXT:
**TXT @PAPER/-SP


*********  RUN STATISTICS **********
   922 RECORDS READ   895 RECORDS WRITTEN   23 PAGES GENERATED

      102 RECORDS USED IN TEMP FILE.
ONE OR MORE "SAVED STATE" RECORDS REMAIN STACKED.

MULTIPLE INPUT FILES USED:
                abstract.txt
                intro.txt
                standard.txt
                validate.txt
                swedrpt.txt
                omsirpt.txt
                compare.txt
                conclude.txt
                ref.txt

*******************************************************************************
```

Open Forum For Members

SPERRY UNIVAC

P.O. Box 43942 MS-4162
St. Paul, Minnesota 55164

Rick,

With all this talk about Ada replacing Pascal as the avant-garde language of the eighties, I thought I would contribute these definitions from *The Name for Your Baby*, by Jane Wells and Cheryl Adkins [Westover Publishing Company, Richmond, Virginia. 1972]:

ADA: (Aida, Eng.) "Prosperous, happy"; Old English
PASCAL: Born of suffering; Hebrew

But then again, what's in a name?

Scott H. Costello

MATHEMATISCHES INSTITUT
DER LUDWIG-MAXIMILIANS-UNIVERSITÄT
MÜNCHEN
Prof. Dr. Günther Kraus

D 8 MÜNCHEN 2, DEN
THERESIENSTRASSE 39
TEL.: DURCHWAHL 28 94 /
(VERMITTLUNG 2 80 41)

I am going to develop PASCAL - programs for use in pure mathematics (Complex Analytic Geometry, Algebraic Geometry, Algebraic Topology). Who is interested to join ideas and experiences? I am interested in commercial applications, too.

Günther Kraus, Mathematisches Institut der Universität München, Theresienstraße 39, D-8000 München 2 (West Germany)

שי מיקרו מחשבים בע"מ.
גבעת שאול ב', ירושלים; טל: 521111
ת.ד. 3405

SHAI MICRO COMPUTERS LTD.
JERUSALEM, ISRAEL, GIVAT SHAUL B', TEL. 521111, P.O.B 3405
CABLES: RIMCO, TELEX: 25387

Pascal Users Group
DEC
5775 Peachtree
Dunwoody Road
Atlanta, GA 30342

Sirs:

Our firm has developed a Pascal based program generator called "MINIAC" which makes possible an 80-90% reduction in the time required to write typical business data processing programs.

I enclose a brochure describing MINIAC, which we have implemented in the UCSD p-System, a microcomputer environment. We are planning a CP/M implementation soon, and we forsee no special problems in implementing MINIAC in any environment which provides a sufficiently powerful Pascal.

We have been using MINIAC for nine months to develop software for our clients in Israel, and we feel that our initial expectations were fully justified.

We are planning to market MINIAC in the United States, and it is for this reason that we are contacting you. Perhaps MINIAC would be of interest to some of your members.

If so, we would be pleased to answer any questions they may have.

Thanking you in advance for your consideration, I remain

Sincerely,

Alex Ragen
General Manager

ar/hs
enc

303-777-3638

southwest decision systems, inc.
30 west bayaud, suite 201
denver, colorado 80223

(text of notice for Pascal News)

Southwest Decision Systems, Inc. is a small software house in Denver, Colorado, specializing in the writing and installation of Pascal-based software on microcomputers. We would welcome leads from university faculty, in the U.S. or elsewhere, concerning exceptional students near the M.S. (or equivalent) who might be suitable for positions with S.D.S. starting late 1982. Demonstrated ability to conceive and complete a substantial Pascal programming project to a very high standard will be the principal requisite. Replies (from faculty only, please) to David P. Babcock, Southwest Decision Systems, Inc., 30 West Bayaud Avenue, Suite 201, Denver, Colorado 80223.

Comment on A.H.J. Sale's Proposal to Extend Pascal

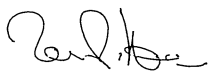
by Tom Pittman
P.O. Box 6539
San Jose, CA 95150

ref: SIGPLAN 16:4 p98-103

It seems to me that while the while-statement and the repeat-statement are "similar" when considered through the flow chart paradigm, they actually have significant differences, resulting (for example) in the fact that dominator analysis requires only one pass if the only loop structure is repeat, but as many passes as the deepest nesting of loops if while-loops are used.

The point is that the repeat-statement performs a valuable service in clearly representing a loop structure that is to be performed one or more times and terminated on a condition generated by the execution of the body of the loop. It is significant that Mr. Sale proposes to filter existing programs by replacing the simple repeat-statement with either a duplication of the body (offering opportunities to have differing versions of the code intended to be the same) or the introduction of that dreaded goto. The repeat-statement cannot be correctly simplified.

Now, I will grant that the repeat-statement may be easily misunderstood. The goto-statement which is offered to replace it is surely no less misunderstood! Merely the fact that neither Mr. Sale's students nor the poor anonymous programmer whose code he set up for us to ridicule are able to grasp the proper distinction between repeat and while, is a poor excuse indeed for the removal of that function from the language. The problem in understanding that gives rise both to the ill-conceived scanner and the terminal I/O excerpt is one of not fully thinking through the program flow, and such a fault will result in incorrect code whether or not the repeat-statement is available to be the butt of misdirected ridicule.



COMPUTACIONES INFOTEC S.R.L.
APARTADO 61125, CARACAS 1060A, VENEZUELA

AV. FRANCISCO DE MIRANDA, GALERIAS MIRANDA, 3º PISO, CHACAO
TELF.: (02) 333590 TLX: 23327 CENINVE

Mr. Dick Shaw,
Pascal User's Group,
P.O. BOX 000524,
Atlanta, Georgia 30338,
U.S.A.

Dear Mr. Dick Shaw:

I receved the ALL-PURPOSE COUPON and I am very interested in joining the group. I am a Software Engineer and our Company INFOTEC is representing microcomputer equipment in Venezuela like ALTOS, TVI, AMADEX, MICROPRO, etc. All our Software is developed in PASCAL (UCSD, PASCAL/M, PASCAL/MT+). Our computers our Z-80 based.

I will submit in the future some ideas or articles concerning our experience in PASCAL. We have developed a General Purpose Data Base Management System Generator. It is Hierarchical and it is only necesary to generate the Schema and all the rest of the system will work. It includes a Data Base Editor for data entry, viewing and editing, General purpose query system used to produce sub-sets of the whole data base, tables of information, reports, etc. Tha tables can de manipulated with our Table System for merging, sorting, joining, and statistical analisis can be carried out with our Stat Package. For the Schema generation there are several programs: Schema editor, list, CRT and Printer format editor, etc.

The system was first developed in UCSD PASCAL but has been transfered to PASCAL/MT+ running on CP/M V2.xx, MP/M V1.xx, ect. It is now a complete menu driven system.

As to the membership you will find enclosed a check for USD 25.00 for a 3 year subscription. Please hurry me the issues.

Sincerly yours,

William Palemen
President
Computaciones INFOTEC, S.R.L.

Pascal Users' Group, c/o Rick Shaw
Digital Equipment Corporation
5775 Peachtree Dunwoody Road
Atlanta, GA, 30342

Dear Mr. Shaw,

For users of interactive systems a very simple modification of the program, Referencer, by Arthur Sale adds a very useful feature. This feature causes all declaration parts to be printed out and thus provides a very handy reference document when developing large programs.

The modification inserts the following:

After line 0785	printflag:=false;
After line 0897	printflaf:=false;
Line 609	remove
Line 610	remove

Sincerely yours,

Edgar S. Gilchrist
218 Via Ithaca
Newport Beach, CA, 92663

Note: My system is AppleII+ and UCSD Pascal.

TRS-80 UCSD PASCAL by FMG

I would be interested in corresponding with anyone who is currently using the UCSD PASCAL package modified for the TRS-80 by FMG Corporation. I have been using the system for personal projects for over a year and am very satisfied with its capabilities, except for one problem which I hope someone else has encountered and solved!!! Programs which utilize random access files (using GET and PUT) appear to randomly destroy blocks on the diskette in the write mode (using PUT). It seems that a bug in the code permits (random) overwrite of some of the diskette sector control information, so that the sector is no longer able to be found. If anyone else has experienced this problem, please get in touch (especially) if you have fixed it. If a P-code disassembler is available for this UCSD PASCAL, I would be very interested in getting a hold of it.

Richard J. Bonneau

6 Tanglewood Drive

Shrewsbury, MA 01545

(617) 845-1432

Pascal Standard: Progress Report

by Jim Miner (1981-07-31)

The second ISO Draft Proposal for Pascal (as printed in Pascal News #20) has received strong support in the official vote this spring. The number of countries disapproving has dropped from four to one.

	Second DP 7185	
Approving	Approving with comments	Disapproving
Italy	Australia	Japan
Netherlands	Austria	
Poland *	Canada	
Switzerland	Czechoslovakia *	
United Kingdom	Finland	
	France	
	Germany	
	United States	

* country is an 'O' member -- vote is advisory.

Some degree of compromise has been reached in the "conformant array parameter" issue (see Pascal News #19, page 74). Because of the convergence of support evidenced by this vote, it is likely that SC5 (the ISO Programming Languages committee) will approve the DP with a few changes at its October meeting in London. Once it has done so, the draft will be a Draft International Standard (DIS) to be voted on by a broader constituency. In short, nearly all of the technical work has been done on the standard, freeing it to progress through the remaining steps toward official adoption. The changes made to the DP will result from the comments submitted by the member bodies with their votes. Tony Addyman and Working Group 4 are presently developing those changes.

The official comments on the DP are quite voluminous, but we have decided to print them here. One reason is that you can get some idea of the amount of effort that goes into each new draft. Remember that these comments are just the output of national committees, and that these committees worked hard to formulate the comments and to reject others. The work done by Tony Addyman at each stage has been tremendous.

Another reason for printing the comments is so you can appreciate the difficulty of some of the technical issues, and the tensions created by conflicting goals of eliminating technical flaws, establishing the standard as quickly as possible, and making the standard as readable as possible. For example, the German comments regarding "denote" raise an issue that pervades the entire document, but its resolution would require many more months and might result in a less readable document.

Finally, note that not everyone is happy with conformant arrays. Both the United States and Japan stress their dislike of including an extension to Niklaus Wirth's Pascal in the first standard. The United States committee is now preparing to put out a draft proposed American National Standard for public comment which will not have any kind of conformant array parameters. Many countries also have criticised certain details of the feature as defined by the second DP; most objected to the use of parentheses in the actual (calling) parameter to specify it as a value (as opposed to "var") parameter. Some changes will therefore be made in the final version.

american national standards institute, inc.
1430 broadway, new york, n.y. 10018
(212) 354-3300

ISO/TC 97/SC 5 N 606
1981 May 08

I S O
INTERNATIONAL ORGANIZATION FOR STANDARDIZATION
ORGANISATION INTERNATIONALE DE NORMALISATION

ISO/TC 97/SC 5

PROGRAMMING LANGUAGES

Secretariat: USA (ANSI)

Summary of Voting on 97/5 N 595 -
Second DP 7185 - Specification for the
Computer Programming Language - Pascal

The Secretariat issued this document for voting by 31 March 1981. To date the following votes have been received:

'P' Members approve	4	Italy, Netherlands, Switzerland, United Kingdom
'P' Members approve with comment	7	Australia, Austria, Canada, Finland, France, Germany, United States
'P' Members disapprove	1	Japan
'P' Members not Voting	6	China, Hungary, Norway, Romania, Spain, Sweden
'O' Members approve	1	Poland
'O' Members approve with comment	1	Czechoslovakia

Comments received :

Australia - Attachment A

Austria - Page 35, paragraph (e) (1) first line: Specification instead of specifiecation

Canada - Attachment B

Czechoslovakia - Attachment C

Finland - Attachment D

France - Attachment E

Germany - Attachment F

Japan - Attachment G

USA - Attachment H - 2 parts

DOCUMENT ISO/TC 97/SC 5 N595

ATTACHMENT A

ISO/DP 7185 - Specification for the
Computer Programming Language PASCAL

Comment of Australian Member Body

In recording a vote of approval on the above ISO/DP, the Australian Member Body submits the following comment:

The Australian vote in favour of the adoption of DP7185.1 expresses the view that the conceptual structure and definition of the DP are correct and appropriate for an International Standard, and takes into account the delays that have already arisen in the preparation and approval of a Pascal Standard.

However, examination of the DP has revealed a number of points which are not adequately defined by the text, though the intent is well-understood by those who have worked on this Standard. The following comments therefore represent our considered view of the editorial changes that must be made to the Draft Proposal so that it does say what is meant. We believe that the changes will be non-controversial, and should be incorporated before the DP is sent for voting as a DIS. Generally the changes correct grammatical and punctuation errors, poor English expression, or omissions.

POSITIVE COMMENT

Comment received on such documents is usually negative, since critical appraisal is sought. It should, however, be placed on record that comments received by the Australian Committee have praised two features of the definition which have raised controversy in the past:

* the conformant-array-parameter, and
* the restriction of a for-statement controlled-variable to local simple variables.

In addition, the improved formalism of the Draft Proposal was favourably received, and the view has been expressed that an even greater use of formal definitions would have been welcome.

TYPOGRAPHICAL COMMENT

PROBLEM
Australia draws attention to the poor presentation of DP7185, and in particular to the following features of the document:

* The typefont (which is guessed to be that of a Decwriter) is very difficult to read in large quantities; its treatment of characters with descenders (for example p, q) is unacceptable in a professional document.

* No underlining or italicising is used in the document, not even where such treatment would aid clarity by giving cues. Thus no headings are underlined or bold-faced, making it difficult to find places in the document. Also, notes should be in a distinctive type-face if possible. Particularly bad examples can be found in section 6.9, where the sense of the words *input* and *output* are only determinable with difficulty:

  DP7185: ...applied to the required textfile output.
  better: ...applied to the required textfile denoted by the required identifier *output*
  or: ...applied to the required textfile *output*.

RECOMMENDATION
While sympathising with the problems associated with the preparation of this document, it is recommended that before the DP is sent out for a further vote, or for voting as a DIS, it should either be typeset or it should be typed with an acceptable word-processing system providing for good-quality typefonts.

INTRODUCTION & ZERO-NUMBERING

PROBLEM
It is barbarous to start the numbering of sections in this document from zero, and offends against normal practice.

In addition, the Introduction is nothing of the sort, but rather part of the prescription of section 1 (Scope of this Standard).

RECOMMENDATION
Delete the "0. INTRODUCTION" heading.

Move the text contained in the now deleted Introduction to the end of paragraph 1.1, page 2.

ERRORS

PROBLEM
The definition of *error* in section 3.1, page 3, is correct, but suffers from two defects. Firstly, the detection of errors is hardly to be regarded as "optional" in accepted English usage; rather the detection of errors may be elided by implementations which do not profess to offer the highest quality of implementation. Unless the meaning is expressed correctly, implementors will take the words in the most relaxing sense.

The second flaw is more serious: the philosophy of errors is nowhere stated. This is certain to cause confusion in future revisions of the Standard, and has been illustrated with the rapid switching of positions on goto-statements in recent drafts. Clearly this is not part of the Standard, but could be in a NOTE.

RECOMMENDATION
1. Alter the definition of error to:

> *3.1* error. *A violation by a program of a requirement of this standard which a processor is permitted to leave undetected.*

2. Between 3.1 and 3.2 add the following NOTEs:

> *NOTE. If it is possible to construct a program in which the violation or non-violation of a requirement of this Standard requires knowledge of the data read by the program, or of the implementation definition of implementation-defined or implementation-dependent features, then violation of that requirement is classed as an* error. *Processors may detect and report on some violations of the requirement without such knowledge, but there always remain some cases which require execution or simulated execution, or proof procedures with the required knowledge. Requirements which may be verified without such knowledge are not classified as errors.*
>
> *NOTE. Processors should attempt the detection of as many errors as possible, and to as complete a degree as possible. Permission to omit detection is provided for implementations in which the detection would be an excessive burden, or which are not of the highest quality.*

Definition of Processor

PROBLEM
The definition of *processor* is incorrect. A processor can only be regarded as a complete system for processing Pascal programs, and parts of a complete system cannot be regarded as a "processor".

A partial processor (eg a compiler, as suggested by the DP) is free of all sorts of semantic constraints; even with a run-time system it can still shed responsibility to a host operating system, or even to hardware design.

If validation of Pascal processors is to be possible, this definition must say what has been assumed all along: a Pascal processor is an entity that accepts Pascal programs, and "executes" them.

RECOMMENDATION
Replace definition 3.4, page 3, by:

> *3.4* processor. *A system or mechanism which accepts a program as input, prepares it for execution, and executes the process so defined with data to produce results.*
>
> *NOTE. A processor may consist of an interpreter, a compiler and run-time system, or other mechanism, together with an associated host computing machine and operating system, or other mechanism for achieving the same effect. A compiler in itself, for example, does not constitute a processor.*

Required, Predefined & Predeclared

PROBLEM
There are a collection of problems with the terms required, predefined, and predeclared in the DP. These are detailed below.

* The terms predefined and predeclared are not defined in the DP, and are not common English words. Their meaning in the context of the DP is thus uncertain, and only determined by Pascal tradition.

* The term required is defined by 6.2.2.10 and nowhere else. A definition of the meaning of the term is necessary, especially as it does not mean predefined nor predeclared.

* In clause 4 an assumption relating to the denotations of required identifiers in program fragments in the DP is stated, but in terms of "predefined or predeclared". Not only are these not defined, but Pascal tradition would then exclude *input* or *output* from the set.

RECOMMENDATIONS

1. Replace the following sentence in section 4, page 3, lines 18-21:
   Any identifier that is defined in clause 6 as the identifier of a predeclared or predefined entity shall denote that entity by its occurrence in such a program fragment.
   by:
   Any identifier that is defined in clause 6 as a required identifier shall denote the corresponding required entity by its occurrence in such a program fragment.

2. Add at the end of the first paragraph of 6.1.3, page 6:
   Identifiers that are specified to be required *shall have special significance in Pascal (see 6.2.2.10 and 6.10).*

3. Add the following sentence after the last paragraph of section 6.3, page 11:
   The required constant-identifiers are specified in 6.4.2.2 and 6.7.2.2.

4. Replace the sentence following in section 6.4.1, page 12;
   The required types shall be denoted by predefined type-identifiers (see 6.4.2.2 and 6.4.3.5).
   by:
   The required type-identifiers and corresponding required types are specified in 6.4.2.2 and 6.4.3.5.

5. Replace the only paragraph of 6.6.4.1, page 38, by:
   The required procedure-identifiers and function-identifiers and the corresponding required procedures and functions shall be as specified in 6.6.5 and 6.6.6 respectively.

6. Add at the end of section 6.2.2.10; page 10:
   See 6.1.3, 6.4.1 and 6.6.4.1.

   NOTE: The required identifiers input *and* output *are not included, since these denote variables.*

7. Replace the first sentence of the second paragraph of section 6.10, page 65:

The occurrence of the identifier input or the identifier output as a program parameter shall constitute its defining-point for the region that is the program-block as a variable-identifier of the required type denoted by text.

by

The occurrence of the required identifier input *or the required identifier* output *as a program parameter shall constitute its defining-point for the region that is the program-block as a variable-identifier of the required type denoted by the required type-identifier* text.

8. The example at the end of 6.6.2 violates the requirements of section 4 by using the required identifier *new* with a denotation that is not the required procedure. Though the usage is obvious, it is inconsistent, and the example should be rewritten with the identifier *new* replaced by *estimate*.

Language Levels

PROBLEM

The DP defines two "levels" of the language, which it numbers 0 and 1. There are two objections to this scheme:

* Numbering an enumerated set of objects 0 and 1 is a barbarism in the English language, however mathematically attractive it might be. Levels 1 and 2 would be far preferable.

* The level chosen to be level 0 is in fact close to what is popularly known as Standard Pascal, whereas level 1 contains an extension which is at present not common. It would therefore be preferable to refer to the "levels" by names which indicate their usage.

The Australian recommendation is to adopt the latter course, using the names Standard Pascal and Extended Pascal to distinguish the levels. Not only does this make the distinction clear, it has the following advantages:

* Vendors of Pascal products can more readily identify their conformance as being to "Standard Pascal as defined in ISO7185" etc.

* Future revisions of the Standard can retain Standard Pascal as a subset, by confining extensions to Extended Pascal.

* Implementors who choose not to implement the extension for conformant arrays will not be saddled with an implied deficiency ("only level 0").

RECOMMENDATION

Replace the phrases *at level 0* and *at level 1* in section 5.1, page 4, and in section 5.2, page 5 by *as Standard Pascal* and *as Extended Pascal* respectively.

Replace the NOTE in section 5, page 4 by:

NOTE. There are two levels of compliance, known as Standard Pascal *and* Extended Pascal. Standard Pascal *does not include conformant array parameters.* Extended Pascal *does include conformant array parameters.*

Replace the several occurrences of

[*do*] *not apply to level 0*

in sections 6.6.3.6, page 35; 6.6.3.7, page 35; and 6.6.3.8, page 37 (and any other occurrences) by:

[*do*] *not apply to* Standard Pascal

Wherever any further occurrences of levels 0 or 1 appear, replace them by appropriate text; a full cross-reference was not available to us to check that all have been detected.

Detection of Violations

PROBLEM

Section 5.1(e) requires the detection of violations that are not errors. However, it does not require that the detection by the processor be reported to the user of the processor.

Secondly, it is unreasonable for the Standard to insist on processors reporting all violations. Parasitic effects of one error may mask some violations and often do; other processors often have error-limits. Interpreters, of course, adopt a different approach to error-detection. The thinking in this section is confused: the appropriate requirement is that the processor be able to classify programs into two classes:

1. The class of compliant programs, and
2. The class of non-compliant programs.

However, if the processor has not completely examined a program text, as occurs in processors with an error limit, processors which abort under some table overflow conditions, or direct execution or interpreter machines, then a third response is permissible:

8. The class of programs in which no non-compliant feature has yet been detected, but which has not yet been completely examined.

Processors should report accordingly, and this should be the Standard's stance. More information about the source of non-compliance in such programs cannot be legislated for as it is heavily dependent on technique.

RECOMMENDATION

Replace section 5.1(e), page 4, by:

(e) determine whether or not a program violates any requirement of this standard that is not designated an error and report the result of this determination to the user of the processor. In the case where the processor does not examine the whole of the program, the user shall be notified that the determination is incomplete whenever no violations have been detected in the program text examined.

Add a NOTE at the end of Section 5.1, page 5:

NOTE. Normally a processor which consists of a compiler and ancillary components will be able to classify programs into the compliant or non-compliant categories in accordance with clause 5.1(e) after examining the program text. However, in cases where the compilation is aborted due to some limitation of tables, etc, an incomplete determination of the kind "No violations were detected, but the examination is incomplete" will satisfy the requirements of clause 5.1(e). In a similar manner an interpretive or direct execution processor may report an incomplete determination for a program of which all aspects have not been examined.

Error-Reporting

PROBLEM
The requirement stated in section 5.1(f) does not require that all the statements relating to error-reporting be easy to find, and indeed they may be obscurely hidden in an obscure part of the documentation and widely scattered. This is undesirable.

RECOMMENDATION
Add the following to the end of 5.1(f), pages 4 & 5:

> *If any violations that are designated as errors are treated in the manner described in 5.1(f)(1), then a note referencing each such treatment shall appear in a separate section of the accompanying document.*

Restrictions and Compliance of Processors

PROBLEM
Though the DP addresses the problems of specifying extensions in section 5.1(g), nowhere is it stated what action processors must take with respect to restrictions. It is possible to argue that no restrictions are possible, and processors must comply with all requirements of the Standard if they are to claim compliance with it, but Australia considers that this is unrealistic. Processors will contain restrictions, even if only a few.

In addition, ignoring the problem effectively prohibits any new reserved words, since these restrict the set of permissible identifiers, thus encouraging overloading of existing operators, words, and other extension mechanisms.

Australia argues that the DP should contain a statement controlling the use of compliance statements, which specifies action with respect to restrictions.

RECOMMENDATION
Add at the end of section 5.1, page 5, but not dependent on (i), the following:

> *A processor that purports to comply, wholly or partially, with the requirements of this Standard shall do so only in the following terms. A compliance statement may be produced by the processor as a consequence of using the processor, or may be included in accompanying documentation. If the processor complies in all respects with the requirements of this Standard the compliance statement shall be:*
>
> <This processor> complies with the requirements of <Standard Pascal> as stated in ISO7185, 198-.
>
> *If the processor complies with some but not all of the requirements of this Standard then it shall not use the above statement, but shall instead use the following compliance statement:*
>
> <This processor> complies with the requirements of <Standard Pascal> as stated in ISO7185, 198-, with the following exceptions:
> <followed by a reference to, or a complete list of, the requirements of the Standard with which the processor does not comply.>
>
> *In both cases the text* <This processor> *may be replaced by an unambiguous name identifying the processor, and the text* <Standard Pascal> *may be replaced by* Extended Pascal *if appropriate to the level of implementation.*
>
> *NOTE. Processors that do not comply fully with the requirements of the Standard are not required to give full details of their failures to comply in the compliance statement; a brief reference to accompanying documentation which contains a complete list in sufficient detail to identify the defects is sufficient.*

Complying Programs

PROBLEM
The NOTE at the end of section 5.2, page 5, is grossly misleading. The results produced under the conditions stated certainly are required to be the same for a class of programs, while other classes have constraints which permit different results. The resultant confusion requires that the Standard say precisely what is implied, not an incorrect statement.

RECOMMENDATION
Delete the NOTE at the end of 5.2, page 5, and replace it by the following:

> *NOTE. A program that complies with the requirements of this clause may rely on particular implementation-defined values or features, and it may contain errors which will only be evoked by particular data values.*
>
> *NOTE. The requirements for compliant programs and compliant processors do not require that the results produced by a compliant program are always the same when processed by a compliant processor. They may be, or they may differ, or potential errors may be evoked, depending on the program. The simplest program to illustrate this is:*
> *program x(output); begin writeln(maxint div (maxint-32767)) end.*

Character-Strings

PROBLEM
The description of character-strings and the denotation of string-elements in 6.1.7, page 7, is confusing, and omits to give the apostrophe-image a value of char-type, except by implication. Also the term "string of characters" is used in a context where "character-string" is more appropriate.

RECOMMENDATION
Delete the text paragraph in 6.1.7, page 7, and replace by:

> *6.1.7* Character-strings. *A character-string containing a single string-element shall denote a value of the required char-type (see 6.4.2.2). A character-string containing more than one string-element shall denote a value of a string-type (see 6.4.3.2) with the same number of components as the character-string contains string-elements. Each string-element shall denote an implementation-defined value of the required char-type, subject to the restriction that no such value may be denoted by more than one string-element.*
>
> *NOTE. Conventionally, the apostrophe-image is regarded as a substitute for the apostrophe character, which cannot be a string-character.*

SUBSIDIARY NOTE
The required values of char-type are:

the ten digit-values denoted by '0', '1', '2', ... , '9'	6.4.2.2
the space-value denoted by ' '	6.4.3.5
the number-values denoted by '+','-','.'	6.9.4.x
the exponent-value denoted either by 'e' or 'E'	6.9.4.5.x
whatever case letters are required for 'True' and 'False'	6.9.4.6

In the preceding redraft, the value denoted by the apostrophe-image is added as a required value, but it need not denote a value whose graphical representation is indeed the ' character. This is exactly the same situation as exists with the other required values: the external graphical representations of the values are not controlled.

LEXICAL ALTERNATIVES

PROBLEM 1
The second NOTE in section 6.11, page 68, is incorrect. The Standard does indeed exclude the existence of other symbols, since processors which accept them are probably (depending on the symbol) accepting programs which are not compliant Pascal programs, and therefore contain extensions.

RECOMMENDATION
Delete NOTE 2 on page 68, and the numeral "1" from the first NOTE.

PROBLEM 2
This whole section is at variance with section 6.1, which sets out the requirements for lexical tokens. Properly, it belongs there, not here at the end of the Standard, which is simply where Niklaus Wirth put it originally in the User Manual.

RECOMMENDATION
Delete section 6.11 and insert a new section 6.1.9 as follows:

> *6.1.9 Lexical alternatives. The representation for lexical tokens and separators given in sections 6.1.1 to 6.1.8 constitutes a* reference representation *for these tokens and separators which shall be used for program interchange.*
>
> *To facilitate the use of Pascal on processors which have a character set which will not support the reference representation, the following alternatives are provided. All processors which have the required characters in their character set shall provide both the reference representations and the alternative representations, and the corresponding tokens or separators shall not be distinguished.*
>
> *The alternative representations for tokens are given below:*

Reference token	*Alternative token*
^	@
[	(.
]	.)

> *NOTE. The character ↑ which appears in some national variants of the ISO character set is regarded as identical to the character ^.*
>
> *The alternative forms of comment are all forms of comment where one or both of the following substitutions are made:*

Delimiting character	*Alternative delimiting pair of characters*
{	(*
}	*)

> *NOTE. A comment may thus commence with "{" and end with "*)", or commence with "(*" and end with "}".*

IDENTIFIER AND LABEL TERMINOLOGY

PROBLEM
The following problem was drawn to Australia's attention by W.Price, but the solution differs slightly from that proposed. It is however based on the comments received, but modified to cope with labels.

In section 6.2.2 the word *identifier* is used with at least four meanings. The one attached to the syntactic definition should be left untouched, but the others need to be distinguished to clarify the DP. Labels are equally affected.

RECOMMENDATION

1. Change the second sentence of 6.1.3, page 6, to read:
   All characters of an identifier shall be significant in distinguishing between identifiers.

2. Replace clause 6.2.2.5 by:
   When an identifier or label has a defining-point for region A and an identifier or label that cannot be distinguished from it (see 6.1.3 and 6.1.6) has a defining-point for some region B enclosed by A, then region B and all regions enclosed by B shall be excluded from the scope of the defining-point for region A.

3. Replace clause 6.2.2.7 by:
   The scope of a defining-point of an identifier or label shall include no defining-point of another identifier or label that cannot be distinguished from it (see 6.1.3 and 6.1.6).

4. Change
   ...all occurrences of that identifier or label shall be designated applied occurrences...
   in clause 6.2.2.8 to read:
   ...each occurrence of an identifier or label which is indistinguisable from the identifier or label of the defining-point (see 6.1.3 and 6.1.6) shall be designated an applied occurrence of that identifier...

5. Change
   ...a type-identifier may have an applied occurrence in the domain-type...
   in clause 6.2.2.9 to read:
   ...an identifier may have an applied occurrence in the type-identifier of the domain-type...

FUNCTION STYLISTICS

PROBLEM
An example of a procedure-and-function-declaration-part is given in section 6.6.2, pages 31 & 32. Amongst the examples is an example of functions using mutual recursion, and illustrating the *forward* directive. This example is written with poor stylistics, in that:

* the mutuality of the recursion is disguised by the layout, in which the two procedures are written differently;
* Apart from the Standard-oriented comment at the top, the mutuality of the recursive references is not documented; and
* a pseudo-repetition of the parameter list of ReadOperand suggests that this poor practice of repeating information (possibly erroneously) be copied.

RECOMMENDATION
Replace the text beginning "{This example of ..." to the end of the section by:

```
{ The following two functions analyse a parenthesized expression and convert it
to an internal form.  They are declared forward since they are mutually recursive -
they call each other. }
function ReadExpression : formula;
  forward;
function ReadOperand : formula;
  forward;

function ReadExpression;  { See forward declaration of heading. }
  var
    this : formula;
  begin
    this := ReadOperand;
    while IsOperator(nextsym) do
      this := MakeFormula(this, ReadOperator, ReadOperand);
    ReadExpression := this
  end;

function ReadOperand;  { See forward declaration of heading. }
  begin
    if IsOpenParenthesis(nextsym) then
      begin
        SkipSymbol;
        ReadOperand := ReadExpression;
        { nextsym should be a close-parenthesis. }
        SkipSymbol
      end
    else
      ReadOperand := ReadElement
  end;
```

Conformant Array Syntax

PROBLEM

The syntax for index-type-specification does not use bound-identifier.

RECOMMENDATION

Replace the syntax for this in section 6.6.3.7, page 36, lines 16-18, by:

index-type-specification =
bound-identifier ".." bound-identifier
":" ordinal-type-identifier .

For-Statement Specification

PROBLEM
In 6.8.3.9, pages 55 & 56, a circular argument is introduced in following the consequences of making the limit expressions e1 and e2 "compatible" rather than "assignment-compatible" with the control-variable. Firstly, the fourth sentence of the second paragraph states:

> *The value of the final-variable shall be assignment-compatible with the control-variable when the initial-value is assigned to the control-variable.*

Later, the paragraph goes on:

> *Apart from the restrictions imposed by these requirements, the for-statement*
> *for v := e1 to e2 do body*
> *shall be equivalent to*
> ...

and this shows that an over-riding restriction is specified in terms of a subsidiary specification (which is valid only where not in conflict with the previous restrictions). Secondly, the similar restriction on e1 is not mentioned at all, and is only implied by the equivalent program-fragment.

The problem is derived from the decision to abandon "assignment-compatibility" as the prime requirement for the limit expressions under all uses. However, if that decision is left, then it can readily be seen that the proper restriction is related to the execution or not of the controlled statement ("body"), not of components of a (virtual) equivalent fragment, and its execution-sequence.

RECOMMENDATION
Delete the sentence given above (first italicised entry) and replace it by:

> *The initial-value and the final-value shall be assignment compatible with the type of the controlled-variable if the statement of the for-statement is executed.*

Trivial Mistakes

PROBLEM
The DP contains several trivial punctuation and grammatical mistakes.

RECOMMENDATIONS

1. Delete second comma in second sentence of 6.4.4, page 21.

2. Delete comma in NOTE on page 16 of 6.4.3.2.

3. In 6.4.3.4, page 19, line 9, insert the word *type* so that the first sentence of the paragraph begins:
   > *For every ordinal-type S, there exists an unpacked set type designated ...*

4. In 6.4.3.2, page 16, replace *characters* by *string-elements* and *left to right* by *textual* in lines 7 and 8 respectively.

5. In 6.5.1, page 24, line 3, delete the text
   (current)
   or remove the parentheses.

ATTACHMENT B

Canadian Standards Association
Association Canadienne de Normalisation

Rexdale, Ontario

COMMITTEE CORRESPONDENCE

Please address reply to writer at:

Anthony Bickle
Chief
Scientific Computing Division
Data Analysis and Systems Branch
Computing & Applied Statistics Directorate
Place Vincent Massey
Ottawa Canada
K1A 1C7
#(9-997-3522

March 6, 1981

CAC/ISO/TC97/SC5 Position to

CNC/ISO Secretariat Letter

File No. SCC ID 504 (97/5)-2

DP 7185

We approve DP 7185 as presented, though making

the following comments of an editorial nature:

COMMENT ON Error Handling (5.1f)
STATUS Editorial
PROBLEM STATEMENT

Parts 2 and 3 of this section (5.1f) say

'2) the processor shall have reported a prior warning that an occurrence of that error was possible;
3) the processor shall report the error during preparation of the program for execution;'

The term 'prior warning' presumably means a warning prior to execution. That is, this warning occurs during preparation of the program for execution. Rewording part 2 makes it clearer that parts 2 and 3 deal with distinct, but related, issues.

PROPOSED CHANGES

Replace 5.1f part 2 with

'2) the procesor shall report during preparation of the program for execution that an occurrence of that error was possible;'

COMMENT ON Numbers (6.4.2.2)
STATUS Editorial
PROBLEM STATEMENT

This section says 'The values shall be a subset of the whole numbers, denoted as specified in 6.1.5 by the signed-integer values (see also 6.7.2.2).' The values are denoted not by values, but by the syntactic class signed-integer.

PROPOSED CHANGES

In section 6.4.2.2, replace '...by the signed-integer values...' by '...by signed-integer...' and replace '...by the signed-real values.' by '...by signed-real.'.

COMMENT ON File-types (6.4.3.5)
STATUS Error
PROBLEM STATEMENT

In part d of the definition of a sequence-type, the case in which y is empty and x is non-empty is not covered.

PROPOSED CHANGES

Replace

'If x is the empty sequence, then x=y shall be true if and only if y is also the empty sequence.'

with

'If either x or y is the empty sequence, then x=y shall be true if and only if both x and y are empty;'

COMMENT ON Example in 6.6.2
STATUS Program Bug
PROBLEM STATEMENT

In function ReadExpression, the statement

```
'this := MakeFormula (this, ReadOperator, ReadOperand);'
```

would not be standard-conforming if both ReadOperator and ReadOperand were functions that advance the input stream - it relies on the left-to-right evaluation of the actual parameters.

PROPOSED CHANGES

Replace 'function ReadExpression ... end;' with

```
'function ReadExpression : formula;
    var
      this : formula;
      op : operator;
    begin
      this := ReadOperand;
      while IsOperator (nextsym) do begin
        op := ReadOperator;
        this := MakeFormula (this, op, ReadOperand);
      end;
      ReadExpression := this
    end;'
```

COMMENT ON Actual parameters with packed types (New 6.6.3.1 and 6.6.3.7)
STATUS Editorial
PROBLEM STATEMENT

Does the sentence

> 'An actual variable parameter shall not denote a component of a variable that possesses a type that is designated packed.'

mean that the component's type must not be packed, or that the variable's type must not be packed? The latter interpretation is the desired one.

PROPOSED CHANGES

In 6.6.3.1 replace the ambiguous sentence with

> 'An actual variable parameter shall not denote a component of a variable where that variable possesses a type which is designated packed.'

Similarly, in 6.6.3.7 replace

> '...shall not denote a component of a variable that possesses a type that is designated packed.'

with

> '...shall not denote a component of a variable where that variable posseses a type which is designated packed.'

COMMENT ON Conformant array parameters (New 6.6.3.7)
STATUS Editorial
PROBLEM STATEMENT

This section (6.6.3.7) says

> '...and which shall have a component-type that shall be that denoted by the type-identifier contained by the conformant-array-schema in the conformant-array-parameter-specification and which shall have the index-types of the type possessed by the actual-parameters that correspond (see 6.6.3.8) to the index-type-specifications contained by the conformant-array-schema in the conformant-array-parameter-specification.'

Since Pascal does not have true multi-dimension arrays, the sentence should be phrased in terms of nested conformant array schemas.

PROPOSED CHANGES

Replace the sentence tail quoted above with

> '...and which shall have a component-type that shall be that denoted by the type-identifier or conformant-array-schema closest-contained by the conformant-array-parameter-specification and which shall have the index-type possessed by the actual parameters that correspond (see 6.6.3.8) to the single index-type-specification closest-contained by the conformant-array-schema in the conformant-array-parameter-specification.'

As is the case elsewhere, this definition applies to the long-hand form of conformant-array-parameter-specifications.

COMMENT ON Assigning-reference (6.5.1)
STATUS Error
PROBLEM STATEMENT

The definition of assigning-reference in section 6.5.1 does not say anything about actual parameters to required procedures other than read and readln. As it turns out, there is no real need since the notion of assigning-reference is only used in the definition of the for-statement, and the type of the loop variable cannot be an array-, pointer-, or file-type. The term 'assigning-reference' and its placement in 6.5.1 give one the misleading impression that it is a generally useful notion.

PROPOSED CHANGES

If the term assigning-reference is to remain specific to ordinal-types then either a) change the name to 'ordinal-assigning-reference', or b) move the definition (6.5.1) to 6.8.3.9 (for-statements).

If the term is to be made generally useful, then to the definition of assigning-reference, append

'(g) The variable is denoted by the variable-access in a procedure-statement that specifies the activation of the required procedure new.

(h) The variable is denoted by the third actual parameter in a procedure-statement that specifies the activation of the required procedure pack.

(i) The variable is denoted by the second actual parameter in a procedure-statement that specifies the activation of the required procedure unpack.

(j) The variable is denoted (possibly implicitly) by the file-type actual parameter in a procedure-statement that specifies the activation of any of the following required procedures: read, readln, write, writeln, get, put, reset, rewrite, and page.

NOTE: It is possible for a processor to determine all assigning-references in a statement without having to execute the program. It is used in the definition of the for-statement.'

COMMENT ON Implementation-Dependencies v.s. Extensions
STATUS Error
PROBLEM STATEMENT

The standard is confused with respect to the nature and varieties of implementation-dependencies. We propose the following characterizations of the terms 'implementation-dependent' and 'extension'.

An 'implementation-dependent' aspect of the language is one for which the standard does not give a complete definition. The intention is to allow the implementor a greater degree of freedom than is normally the case. The following characteristics are desireable:

1) A standard-conforming processor may choose any implementation of an implementation-dependent feature as long as it meets the requirements set down by the standard.

2) A standard-conforming processor need not document the way(s) in which the implementation-dependent aspects of the language are implemented (c.f. implementation-defined aspects).

3) A standard-conforming program may not rely on the manner in which an implementation-dependent aspect is implemented.

On the other hand, the term 'extensions' is used for '...any features accepted by the processor that are not specified in clause 6.' The intention of talking about extensions in the standard is to allow an implementation to augment the language defined in the standard. Extensions have the following characteristics:

1) Standard-conforming processors may support extensions.

2) Standard-conforming processors must be able to process the use of any extensions '...in a manner similar to that specified for errors...'

3) Standard-conforming processors must document all extensions.

4) Standard-conforming programs must not use any extensions.

PROPOSED CHANGES

It would seem appropriate to define the term extensions in section 3 instead of in section 5.1 by adding

'3.5 extension. A feature accepted by a processor that is not specified in clause 6.'

In section 5.1, we find

'(i) be able to process in a manner similar to that specified for errors any use of an implementation-dependent feature.'

This clause in meaningless: any program containing an assignment statement can be said to use an implementation-dependent feature. The violation is in relying on a particular implementation of an implementation-dependent feature. Since detection of such violations is impossible in general, clause 5.1 (i) should be deleted.

A better wording for 5.2 (c) is

'(c) not rely on any particular interpretation of implementation-dependent aspects of the language concomitant with the program's compliance level.'

Section 6.1.4 talks about implementation-dependent directives. Calling such directives implementation-dependent is incorrect - the implementor would not even have to document them! These are extensions - and the standard has adequate constraints on extensions. Therefore, delete the sentence .'Other implementation-dependent directives may be provided.' and change

'NOTE: On many processors the directive external is used to specify that the ...'

to

'NOTE: Many processors provide, as an extension, the directive external which is used to specify that ...'

The implementation-dependencies mentioned in sections 6.7.2.1, 6.7.3, 6.8.2.2, and 6.8.2.3, are true implementation-dependencies - no changes are needed.

As suggested in another comment, the effect of inspecting a textfile to which page have been applied should be implementation-defined, not implementation-dependent.

In section 6.10 we find

'The binding of the variables denoted by the program parameters to entities external to the program shall be implementation-dependent, except if the variable possesses a file-type in which case the binding shall be implementation-defined.'

As is the case with directives, we don't want the implementor going off and providing non-file-type program parameters without documenting them; this should be called an extension. Replace the above sentence with

'The variables denoted by the program parameters shall possess a file-type and the binding of the variables to entities external to the program shall be implementation-defined.'

If it is still deemed necessary to mention the common extension, extend the note as follows:

'NOTE: The external representation of such external entities is not defined in this standard, nor is any property of a Pascal program dependent on such representation. As an extension, many processors permit the variables denoted by the program parameters to possess a type other than a file-type.'

COMMENT ON If statements (6.8.3.4)
STATUS Editorial
PROBLEM STATEMENT

This section says 'An if-statement without an else-part shall not be followed by the token else.' It is only a problem if an if-statement without an else-part is IMMEDIATELY followed by the token else.

PROPOSED CHANGES

Change the sentence to read: 'An if-statement without an else-part shall not be immediately followed by the token else.'

COMMENT ON Procedure page (6.9.6)
STATUS Editorial
PROBLEM STATEMENT

'The effect of inspecting a textfile to which the page procedure was applied during generation shall be implementation-dependent.' It would be more appropriate if this aspect was implementation-defined, not implementation-dependent. This would also be consistent with stance taken in 6.10 where the effect of the application of reset or rewrite to either input or output was classed as implementation-defined.

PROPOSED CHANGES

Change the sentence to: 'The effect of inspecting ... shall be implementation-defined.'

COMMENT ON Terminating execution of programs (5.1 i 3)
STATUS Editorial
PROBLEM STATEMENT

Section 5.1, part i, subpart 3 says 'the processor shall report the error during execution of the program, and terminate execution of the program.' An implementation should be free to decide (and document) what form of corrective action, if any, will be taken in the event of a runtime-detected problem. For example, the processor might want to ask the user what value his uninitialized variable should have, and then resume execution.

PROPOSED CHANGES

Change the sentence to: '4) the processor shall report the error during execution of the program.'

Comment on Value Conformant Arrays (6.6.3.7)

Status: technical comment

Problem Statement:

An actual-parameter corresponding to a conformant-array-parameter-specification is allowed to be an expression (that is not a variable-access). This results in copying of the value of the actual-parameter.

This approach is conceptually inappropriate, inconsistent with the rest of the language, and error-prone. In PASCAL, it has been the programmer of the procedure declaration who has decided (by choosing between the variable and value forms of formal parameter specifications) whether a local copy of an actual-parameter is necessary. This responsibility should not fall on the callers of a procedure because, in principle, they need only concern themselves with what the procedure does, and should not be concerned with how this is done. If parenthesization of an actual conformant array parameter is by accident omitted, the result will often be a subtle logical error because of unexpected storage sharing, with no compile-time or run-time warning.

Proposed Changes:

1. Allow value as well as variable forms of conformant-array-parameter-specifications.

2. Require an actual-parameter which corresponds to a variable conformant-array-parameter-specification to be a varriable-access.

3. Modify the restriction in the last paragraph of 6.6.3.7 to apply only when the actual-parameter corresponds to a value conformant-array-parameter-specification.

ATTACHMENT C

Czechoslovak comments of an editorial nature on document ISO/TC 97/SC 5 N 595 - DP 7185

1) In our opinion, the incorporation of levels 0 and 1 into the specification of Pascal in fact defines two programming languages, being inconsistent with the need of portability of programs.
We suggest therefore to retain one level of compliance only, preferably level 1 (including conformant array schema) to force compiler producers to include this required feature into their products.

2) In section 6.4.3.4 a statement limiting the largest and smallest values of the base-type was deleted. We are convinced that such limits exist in each implementation and are usually low.
We suggest to add a statement to section 6.4.3.4, stating an existence of limits of the cardinality of canonical sets (these limits being implementation-defined) and requiring their minimal range to allow for set of char.

3) The behaviour of the procedures read and readln is not satisfactorily resolved when reading integer- or real-type values.
We suggest to adjust parts (c) and (d) of section 6.9.2 in such a way, that if rest of file being scanned for integer or real values consists of spaces and end-of-lines only, then reading shall cease, eof and eoln being true and value of variable v being left undefined.

4) The production rule for procedure-statement conflicts with the definition of parameter-lists for procedures read, readln, write, writeln.
We suggest to formally complete the production rule for procedure-statement as follows:

```
procedure-statement =
    procedure-identifier [actual-parameter-list]      /
    read-procedure-identifier  read-parameter-list    /
    readln-procedure-identifier  readln-parameter-list    /
    write-procedure-identifier  write-parameter-list  /
    writeln-procedure-identifier  writeln-parameter-list .
```

ATTACHMENT D

COMMENTS OF SFS ON DP7185 "SPECIFICATION FOR THE PROGRAMMING LANGUAGE PASCAL"

Finnish comments are mainly based on the paper prepared at the Helsinki University of Technology and made by the PAX-Pascal Group (Jukka Korpela, Pertti Tapola, Timo Larmela, Ahti Planman). I have collected some other opinions listed below.

Layout of the draft is incomplete: It's very difficult to find starting points oh chapters from the text, because there are no extra emty lines between chapters, Darker chapter headings or headings written with letters differing from normal text would help. Contents (page 1) is incomplete and doesn't include all chapter headings. Index (pages 77-82) is very uncomfortable to use because of several references to same objects (for example term "variable" has 23 references). References should be grouped into "sub-terms" or/and main references should be underlined or written with different type. Some terms (for example "comment") are missing.

In chapter 6.1.2 characters "{" and "}" are missing from the production special-symbol. It would also be usefull to have reference to the chapter 6.11 (Hardware representation) where alternative symbols are listed.

In chapters 6.1.8, 6.4.3.1.2 and 6.5.3.2 references to chapter 6.11 as above. Thats important for scandinavian Pascal users, because we use scandinavian letters Ä,ö,Å having same code as [,\,]. Just a few terminals have characters { and } .

In chapter 6.4.3.1 order of productions is wrong, in some other chapters too.

In chapter 6.4.3.3 (record type variant part) it should be possible to have as an element of case-constant-list some kind of subrange expression of form case-constant ".." case-constant. Same form is also usefull in case-statement (6.8.3.5). In addition this form of case-constant is compatible with set expressions.

Basic principles of garbage collection system should be formulaten in spite of it's hardware-dependence. Thats important because different implementations have different properties (e.g. what to do with dynamic allocated variables referenced with pointers written into file-variable.

Tampere 1981-03-16

Hannu Jaakkola
Acting member of SFS on the area of ISO TC97/SC5

HELSINKI UNIVERSITY OF TECHNOLOGY
Computing Centre
PAX-Pascal Group/Jukka Korpela 4-MAR-1981 1(19)

COMMENTS ON THE 2ND DRAFT PROPOSAL FOR THE ISO SPECIFICATION FOR THE COMPUTER PROGRAMMING LANGUAGE PASCAL

CONTENTS

Foreword

This paper has been prepared at the Helsinki University of Technology Computing Centre. It does not present any official statement of any organization but reflects the observations, suggestions, and opinions of several specialists actively working on the fields of systems and applications programming, including Pascal compiler writing and maintenance, and teaching of Pascal.

CHAPTER 1

STRUCTURE AND TERMINOLOGY

1.1 OVERALL STRUCTURE AND COMPLETENESS OF THE DRAFT

The draft being commented contains significant improvements to the first draft, and is, in general, sufficiently complete and well-structured to become a standard.

The main disadvantage is the alteration of terminology and style for semi-formal definitions. This draft, as well as the first draft, contains a great amount of terminology which is not commonly known and used in the Pascal community, or even differs from the terminology currently in use.

For example, the definitions in clause 6.2.3 are difficult to understand, and assumably extremely obscure to ordinary Pascal programmers. What makes them strange for experts too is the obvious attempt to avoid references to implementation. The definitions become understandable to a compiler writer when the "within" relation is conceptually associated with what is known as static link in implementations.

On the other hand, the last note in clause 6.6.3.7 makes a rather explicit reference to implementation, using the notion of activation record.

It is difficult to define some features of Pascal in a manner which is both general (not referring to a particular method of implementation) and understandable, and possibly the difficulty is inherent.

In spite of the criticism above, the difficulties of specification should not be allowed to postpone the standardization of Pascal. Probably a sufficient solution would be to add a few notes referring to implementation aspects, particularly to clause 6.2.3 but possibly also to clauses 6.6.6.3 (about the fact that in practise the address of an actual variable parameter is passed and all references to the formal parameter use the address passed), 6.6.3.4 and 6.6.3.5 (an analogous note would be useful), 6.6.3.7 (e.g. that both the address of an actual parameter and the actual index bounds are passed), 6.8.2.4 (a nonlocal GOTO requires an appropriate context switching), and 6.8.3.10 (the address of a record variable in the record variable list of a WITH statement is calculated once only).

The structure of the draft is similar to previous descriptions of Pascal. However, the order of presentation should be reconsidered in the following respects.

1. Clause 6.3 bears the title Constant-definitions, although it also describes constants. Splitting it into two parts would not be worth while, but the title should be changed.

2. Similar comment applies to clause 6.4. However, the importance of the subject and the length of the clause suggest that the clause should be divided into several major sub-clauses of clause 6. At present clause 6.4 describes type definitions, denotations of types, and the meanings of type denotations. These subjects should be treated separately.

3. Rules for procedure and function declarations in clause 6.6 exhibit great similarity of structure. Integration of the specifications would increase readability and reduce the size of the standard.

1.2 THE STABILITY OF PASCAL

The two major changes stated in the foreword are useful. The first one is to be regarded as a necessary language change. The second one is rather strong extension to the language defined by Niklaus Wirth but is very useful. The solution adopted, to make it a sort of "recommended extension", is elegant.

They are some features of Pascal in which the draft differs from Wirth's definition and/or most current implementations in a manner which makes them important for ordinary users. Mentioning them in the foreword would be worth while. This applies in particular to type compatibility rules in the broad sense, the semantics of WITH statement, the meaning of IN operator, and the format of output of real values to a textfile. The changes involved are definitely improvements.

The definition of Pascal should not be changed from that given in the draft in any essential respect. There are, however, some features which should be specified more exactly.

Moreover, after the official approval of the standard by ISO, a project should be started in order to define "level 2 Pascal", i.e. to standardize some extensions to the language described by the document being currently prepared. It is well known that there are several extensions to Pascal in existing implementations. Often the extensions serve similar purposes but differ in their syntax and/or details of semantics. Given that extensions are available and are used, portability of programs could be increased if the most common extensions were standardized.

The project suggested would inevitably encounter serious problems because of the varying needs of the users as well as the different opinions of language implementors and computer scientists. Anyhow, the Pascal language was designed for teaching - and is undoubtedly the best language for that purpose - but is being used for the construction of complicated "real-life" programs and systems as well. The true applications of Pascal require carefully selected and defined extensions to the language.

Admittedly, Ada is an extension of Pascal, but in roughly the same sense as Pascal is an extension of Algol 60, i.e. very far from being a pure extension. A fundamental difference between Ada and Pascal is that Pascal can be learned in toto within reasonable time, even by a person with no previous experience about computers, whereas Ada is "everything for everybody" which makes the language conceptually difficult and large in contents.

There is no need to suggest what the "level 2 Pascal" would contain. Instead the problem is to limit the extensions to a conceptually clear repertoire which increases the expressive power of the language without substantially decreasing efficiency of implementation. In our opinion, the following extensions (possibly together with some minor extensions) would constitute such a repertoire:

1. Use of static expressions instead of constants.

2. Some kind of module structure.

3. Separate compilation of modules, together with the definition of the properties of the software support needed.

4. Dynamic arrays, which could be added to the language simply by allowing the use of a parameter of a procedure or function in the same manner as constant identifiers in type definitions.

5. Double-precision real numbers.

6. The LOOP EXIT construct.

7. OTHERS branch and/or subrange notation for case constant lists in CASE statement.

8. Additional predefined procedures and functions for file operations (close, delete, append, etc.), including tools for control over input errors like invalid format of numeric data.

9. Standardization of the feature that program parameters declared as array variables represent external random access files.

1.3 CONCEPTS AND DENOTATIONS

When describing a programming language, clear distinction should be made between an underlying concept (an abstract entity) like a variable, and its denotation like a variable denotation. The draft is incomplete in this respect. For variables, such distinction is made in most contexts; but for types not. Moreover, the production rule for variable denotation ("variable-access" in the draft) uses terms like "entire-variable"; a more adequate term would be "entire-variable-denotation".

Consider, for example, clause 6.4.3.5. It first specifies "file-type" by a production, i.e. defines the term "file-type" as one form of type denotation. However, the text then uses the term "file-type" as being something which can be denoted by a type-denoter. Such confusions could be avoided by the systematic distinction mentioned.

1.4 THE STRUCTURE OF LANGUAGE DEFINITION

The draft uses the verb "shall" excessively. A standard, by its very nature, says how things shall (or should) be; undiscriminated use of "shall" is redundant.

Moreover, excessive use of "shall" hides the fact that the different statements in the draft standard have varying logical status. Language definitions (excluding experimental formalized systems) in general consist of (a) rules for context free syntax, usually given in BNF form, (b) additional syntactic rules, given in prose, and (c) semantic rules, given in prose and being somewhat less exact than syntactic rules. The draft uses "shall" both in class (b) and in class (c) rules. It would be more natural to restrict the use of "shall" to class (b) rules, class (c) rules just stating what IS the meaning of a language construct.

In addition, there are the specifications for error conditions, with the word "error" used to designate what is commonly known as runtime error. (A processor may of course be able to detect a runtime error during compilation, in special cases.) In these specifications, "shall" is not necessarily strange but useless.

Yet another group of statements in language definition consists of nominal definitions (for auxiliary concepts). In a sense, a language standard as such is a nominal definition of a language. From the reader's point of view at least, it would be very useful to separate nominal definitions (in the strict sense) from the other contents of the standard. They neither describe the language nor set any requirements upon complying programs or processors, but serve for the purpose of description and specifying requirements.

Consequently, the lowest level clauses of the standard (i.e. clauses not containing any other clause) should be organized as follows. First the relevant production rules are given (in BNF). Then the additional syntactic requirements are specified, in prosè, but exactly, using whatever auxiliary technical terms are needed. Next, the semantic rules are given, in prose, and this specification is sometimes unavoidably inexact (but uniquely interpretable by experienced benevolent readers). Finally, the error conditions, if any, are specified.

Whether the suggested structuring is reflected by appropriate sub-titles, paragraphing, layout, or similar methods, is a matter of convenience. In most cases, paragraphing seems to be the most adequate method. The first prose paragraphs (syntactic rules) may well use the word "shall", whilst the others should use "is".

Nominal definitions should, if possible, be collected into separate clauses, and clearly distinguished as such, e.g. by beginning them with "Definition." or "Convention.". Then it would be unnecessary to use clumsy constructs in English; instead of "a shall be designated as b" one may specify "a is called b", "a is said to be b", or simply "a is b".

To make the suggestions more concrete, here is a revised form of 6.5.5 (with no changes to the contents):

6.5.5 Buffer-variables.

```
buffer-variable = file-variable "^" .
file-variable = variable-access .
```

A file-variable shall be a variable-access that denotes a variable possessing a file-type.

A buffer-variable denotes a variable associated with the variable denoted by the file-variable of the buffer-variable. A buffer-variable associated with a textfile possesses the char-type; otherwise, a buffer-variable possesses the component-type of the file-type possessed by the file-variable of the buffer-variable. A reference or access to a buffer-variable constitutes a reference or access, respectively, to the associated file-variable.

Examples:

```
input^
pooltapeÄ2A^
```

It is an error to alter the value of a file-variable f when a reference to the buffer-variable f^ exists.

The revised form uses the terminology of the draft, and is not to be taken as a final suggestion but rather to illustrate the method of presentation.

The term "implementation-defined" is defined (clause 3.2) too vaguely. In particular, may the corresponding definition (for an implementation) specify additional error conditions, restrictions or even changes to the specifications in the standard?

Especially important problem arises from the fact that binding of program parameters of file type to external entities is "implementation-defined". Does this imply that there must be some binding? If not, it is possible to provide a processor which strictly conforms to the standard but is completely useless. Moreover, it is assumably intended that a program parameter of type Text can be bound to a device like terminal, line printer, or card reader. Now suppose that we bind a such a program parameter, say f, to a terminal, write to the file f, and then try to do Reset(f). Strictly taking this should give us the opportunity to read back what we wrote. (Clause 6.6.5.2 implies that Reset(f) does not change the sequence of components associated with the value of f, except that it may append an end-of-line component to it.) Although this is implementable (by making, say, a disk copy of everything written to f) it pragmatically makes no sense. The problem is even clearer for a file bound to an unspooled card reader, first opened by Reset and then re-opened by Rewrite; since the pre-assertion for Rewrite is True, the operation should definitely be possible. One solution is of course to prevent the binding of a program parameter other than Input or Output to a device; but such a restriction seems unacceptable and it probably is not the intention that the standard would implicitly require it.

Consequently, one should either specify that the definition of an implementation-defined feature introduces modifications to the language specification, or to remove any need for such modifications. (The latter alternative is definitely better, and would require changes to the specification of Reset and Rewrite at least, probably also the specification of real arithmetic operations which should be specified to be an error if the operation is not carried out with sufficient accuracy.)

1.5 TERMINOLOGY

The following changes of terminology are suggested. They would be motivated by the terminology currently in use, or by simplicity, or by a clearer distinction between "things and names", i.e. between (abstract) entities and their denotation.

"The y closest-containing an x" should be replaced by "the smallest y that contains an x". ("Closest-containing" does not correspond to normal rules of formation of words in English.)

"New-type", "new-ordinal-type", etc. should be replaced by "type-description", "ordinal-type-description", etc., "ordinal-type" by "ordinal-type-denotation" (or -denoter), and so on. A type-denotation is a language construct that denotes a type; a type is an abstract entity (and the word "type" as such should be reserved for that purpose); and a type-description is any type-denotation which is not a type-identifier.

Similar changes should be made to terminology related to variables. "Variable-access" should be replaced by "variable-denotation". The variable (as abstract entity) associated with a file-variable should be called "buffer-variable"; it can be denoted by buffer-variable-denotation of the form f^ but it need not (for example, a formal parameter may denote a buffer variable).

"Identified-variable" should be replaced by "referenced-variable" or "referenced-variable-denotation", as appropriate.

The phrase "the type possessed by x" is strange and artificial. It should be replaced by "the type of x". This will be possible when "type" is restricted to refer to an entity, not to a syntactic construct (because "of" applied to syntactic constructs has a specific technical meaning by clause 4).

There seems to be no good reason to use the attribute "required" instead of "predeclared" or "predefined", except that it may shorten some specifications (sometimes "required" should be replaced by "predeclared or predefined"). Admittedly the existence of e.g. the type integer is "required"; but the potential existence of enumerated types is "required" as well. Moreover, the "required" type identifier Integer can appropriately be called "predefined", whereas the type denoted cannot adequately be called "required" or "predefined". It is "introduced by language definition", but such a term would admittedly be clumsy.

CHAPTER 2

DETAILED COMMENTS AND SUGGESTIONS

These comments are organized according to the structure of the draft.

The relevant clauses of the draft are referred by their number only, so that these comments should be read together with the draft.

2.1 LEXICAL TOKENS

The statements "Identifiers may be of any length. All characters of an identifier shall be significant." are redundant and should be made into a note. However, restricting the number of significant characters in identifiers to, say, 10 would not decrease the expressive power of Pascal, would allow compilers to be slightly more efficient, and would promote portability of programs (because in any case programs will be used in environments not supporting infinite recognition length).

The statement "A directive shall occur only in a procedure-declaration or function-declaration." could be misinterpreted so that, for instance, "forward" could not be used as identifier (which is the case in most implementations). A clarifying note should be added.

Clause 6.1.5 states that "An unsigned-real shall denote in decimal notation a value of real-type". The meaning of "denote" in this context requires clarification, since an unsigned-real in general does not exactly correspond to any value of real-type (the internal representation of real numbers being what it usually is). Moreover, it cannot be uniquely derived from 6.1.5 what a processor should do with an unsigned-real whose mathematical value is outside the implemented range. Consider 1e-1000 (assuming a typical floating point representation in which no accurate representation for it exists); should the processor represent the value as 0.0, or as the smallest positive real number representable, or should it give an error message? And what about 1e+1000?

The pseudo-production for string-character should be replaced by a more adequate formulation, e.g. by the following:
The syntax rule for string-character is implementation-defined and shall have the form

string-character = a1 ö a2 ö ... ö aN .

where each of a1, a2, ..., aN is a terminal symbol denoting a single character.

2.2 BLOCKS, SCOPE AND ACTIVATIONS

The draft requires, for a change, that every declared label must be used. Admittedly it is good programming practice not to declare labels which are not used; but why should they be treated differently from identifiers in this respect? A processor may give warnings about unused identifiers and labels or it may not; but to specify such redundancy to be a violation of the rules of language is questionable.

A note should be appended to clause 6.2.2, saying that the scope of an identifier shall not contain applied occurrences of synonymous identifier (from outer scope), if the principle is to remain. However, the proposed scope rules unnecessarily complicate compilers, and it is unlikely that any standard can enforce such rules to be implemented by Pascal processors. We strongly suggest that the definition of scope be revised back to the principle that the scope starts from the defining-point. It would hardly decrease language security, and would be intuitively more understandable than the principle that the scope begins at a point preceding the defining-point.

Clause 6.2.3.2 would be easier to understand if some notes were appended, e.g. a note stating (as in the first draft) that each activation of a block introduces a collection of distinct local variables.

Clause 6.2.3.3 is extremely vague. Is the first statement a nominal definition of "within" relation between activations, or does it prescribe where an activation can be designated (by what?). The statement after the note uses the term "within" to denote a relation between occurrences of labels and identifiers, on one hand, and activations, on the other. Presumably the word "within" should in that context be understood in some intuitively evident sense; but in what sense can an occurrence be within an activation? An occurrence of an identifier primarily appears (textually) within a block, and it obviously denotes some entity which belongs to some activation of that block; but the problem remains: what is the corresponding activation?

2.3 CONSTANT-DEFINITIONS

The semantics of a sign in a constant, however obvious, should be explicitly specified. (Notice that such a sign is not an operator, so that the semantic rules for unary "+" and "-" are not applicable.)

2.4 TYPE-DEFINITIONS

2.4.1 General

The statement "The required types shall be denoted by predefined type-identifiers ---." is redundant.

2.4.2 Simple-types

The alternatives integer-type, Boolean-type, and char-type should be removed from the production for ordinal-type. They are redundant (being special cases of ordinal-type-identifier), there are no productions for them, and the terms are used to refer to the abstract type-entities (instead of identifiers) in the sequel.

The production

real-type = type-identifier .

should be added.

The specification of the required numerical types would be clarified by referring to "the mathematical set of whole numbers" instead of just "whole numbers" and similarly for real numbers.

Since the specification of integer-type in 6.4.2.2 might be interpreted as excluding the possibility of existence of values of that type outside the interval -Maxint..Maxint, it is suggested that the sentence beginning with "The values shall be a subset of the whole numbers ---" be truncated to that part cited; the denotation of values of integer-type by signed-integers is sufficiently described by clauses 6.1.2 and 6.3, provided that the latter is extended by a description of the semantics of a sign, as suggested earlier in these comments.

The rules for subrange-types (in 6.4.2.4) are inexact and given in a confusing order (syntactic requirements being intermixed with semantic specifications). For example, starting the description by "The definition of a type ---" may suggest that subrange type denotations would only be allowed in type definitions, and leaves unspecified what is a definition of a type.

2.4.3 Structured-types

The specification of the effect of PACKED should be made clearer. The phrase "should be economised" can be interpreted so that PACKED is a suggestion only, and the processor may choose not to apply any effective packing even if it would be possible, or a processor may ignore PACKED entirely. This is assumably the intended interpretation; the next paragraph, however, refers to the representation of a type (values of a type) in data storage as being "packed". Evidently this is some confusion, because nothing prevents the processor from representing a structured type not designated packed in a form which is packed (in the sense that minimal storage is used).

Consequently, clause 6.4.3.1 should be modified as follows: First, the only statement that is strictly related to the language definition is made: "The occurrence of the [illegible] packed in a new-structured-type shall designate the type denoted thereby as packed." Then the following is stated in a note: "The designation of a structured type as packed does not designate any component of the type as packed." Then no note about the logical effect is given; this note may read as the note in the draft. Finally, a third note (which is practically very important but logically irrelevant) should be given, e.g. as follows: "On many processors, the designation of a structured-type as packed may cause the representation of values of the type to require less data storage than otherwise would be the case; on the other hand, it may cause operations on, or accesses to components of, values of the type to be less efficient in terms of space, or time, or both."

In 6.4.3.2, as well as in 6.5.3.2 and 6.6.3.7, certain syntactic constructs are defined to be "equivalent". The precise meaning of such definition is left unspecified. What "equivalent" presumably means is roughly what is meant by "identical" according to Leibniz' definition of identity ("Eadem sunt qui inter sibi salva veritate substitui possunt"). Thus, a definition (convention) should be given stating that when two syntactic constructs are defined to be equivalent, this means that either of the two constructs may be replaced by the other without affecting the correctness or meaning of a program, and that any rule given for either construct is applicable to the other as well.

A note should be given in 6.4.3.3, stating that for a variant part without a tag-field, the selector of the variant part does not necessarily have a physical correspondence in the representation of the record type.

Clause 6.4.3.3 allows empty field-lists which implies that an empty record is allowed. However, the question arises whether a variable of an empty record type is initialized or not; on one hand each variable is uninitialized when it comes to existence; on the other hand, a record is initialized when all of its fields are initialized, which means that an empty record would always be initialized. Since empty records are useless, a minor change of definition would remove this theoretical but irritating problem: remove the outermost brackets from the production for field-list, enclose the symbol field-list into brackets in the production for variant, and add (into the text) the requirement that for a field-list with no fixed-part, at least one variant of the variant-part shall contain a field-list.

The draft does not specify any restrictions on the use of ordinal types as the base-type of a set-type. This effectively means that implementation of sets will be rather inefficient, which causes set types to lose a lot of their usefulness. (So this change to the language is an operation which may succeed but the patient may die.) The restrictions, as specified in the first draft, should be restored.

2.5 DECLARATIONS AND DENOTATIONS OF VARIABLES

Clause 6.5.3.2 does not specify the order in which the indices of an indexed variable are evaluated; neither does it state that the order is implementation-dependent. Analogously with e.g. 6.7.2.1, it should be specified that the order of evaluating the index-expressions in an indexed-variable is implementation-dependent.

The production

field-designator-identifier = identifier .

should be included into clause 6.5.3.3.

2.6 PROCEDURE AND FUNCTION DECLARATIONS

Clause 6.6.3.1 specifies that with each formal value or variable parameter there is an associated variable. This specification is somewhat obscure because of the presence of the article "the" (" --- defining-point as the associated variable-identifier ---"). Similar comment applies to procedural and functional parameters. The use of "the" seems to suggest that the existence of such an associated entity has been previously postulated, which is not the case.

Clause 6.6.3 does not specify any restrictions on the allowed types of a formal value parameter. Clause 6.6.3.2 specifies that the actual parameter must be assignment-compatible with the type possessed by the formal parameter. This means that it is legal to declare a procedure with a value parameter of a file type but illegal to call such a procedure. This is somewhat strange; in general, language definition should not formally allow constructs which are useless. The following is suggested:

1. Add the following definition to clause 6.4.3.5, before the first paragraph of the very text: "A type is said to have a file component if it is a file type, or an array type whose component type has a file component, or a record type such that at least one of its fields is of a type that has a file component." Change the paragraph mentioned to read as follows: "The type-denoter of a file-type shall not denote a type that has a file component."

2. Change statement (a) of 6.4.6 to read as follows: "(a) T1 and T2 are the same type which does not have a file component."

3. Add the following sentence to 6.6.3.2: "The type of a formal parameter shall not have a file component."

By 6.6.3.3, "An actual variable parameter shall not denote a component of a variable that possesses a type that is designated packed." However, there is some doubt about the relation of componentship. For clarification, the following note should be added: The relation of componentship is not transitive; that is, if a is a component of b and b is a component of c, then a is not a component of c.

In 6.6.3.7, it is said that the actual parameter corresponding to a conformant array schema "shall be either a variable access or an expression that is not a factor that is not a variable-access". This is not very explicit, and it seems that the contents of that specification is not what is intended: probably the second "not" should be removed? Of course, any variable-access is an expression that is not a factor that is not a variable-access, so the subsequent rules are ambiguous. What is effectively meant is probably that such an actual parameter shall be either a variable or an expression that is either a string constant (possibly in parentheses) or a variable enclosed in parentheses.

On the other hand, the differences between the first draft and the second draft in the specification of conformant-array-schemas clearly show that the authors of the second draft wish to allow conformant-array-schemas as value parameters. We have no strong opinion about such an extension. However, if accepted, the extension should be made in a less confusing way. In general, value and variable parameters are distinguished by the absence or presence of the token VAR in a parameter-specification. We can see no reason why this method should not be used for conformant-array-schemas, too.

The note in clause 6.6.4.1 should not be a note but a part of the very specification of the language. Moreover, it leaves undefined what rules, if any, given for user-declared procedures and functions are applicable to required procedures and functions. This incompleteness is particularly important to the semantics of Write, Writeln, Read, Readln, Pack, and Unpack.

Clause 6.6.5.2 specifies the semantic of Read and Write in terms of an expansion into more primitive statements (cf. also 6.9 for similar expansions). Now if Read(f,a,b) shall be equivalent to BEGIN Read(f,a); Read(f,b) END we have to ask:

1. shall the variable f be evaluated several times
2. shall such evaluation be affected by the effects of the previous operations caused by the statement (consider Read(fÄiÅ,i,j))

Obviously it is intended that access to the file variable is established as the first operation in the execution of the procedures mentioned; this should be specified.

Clause 6.6.5.4 defines the transfer procedures Pack and Unpack as "macros" whose calls must be equivalent to the given expansions. However, it makes no sense to interpret this literally because it would imply that the parameters are name parameters, quite contrary to the nature of the Pascal language. (Literally, 6.6.5.4 would imply that if in, say, Pack(a,i,z), a is an indexed variable (of an array type, of course), its indices should be evaluated N times where N is the number of components of z. Consider the (admittedly theoretical!) possibility that the evaluations of a and z affect each other!) - Thus it should be specified that the parameters of Pack and Unpack shall be evaluated once only, in an implementation-dependent order.

2.7 EXPRESSIONS

Clause 6.7.1 says that "An expression shall denote a value ---", and clause 6.7.2.1 speaks of "evaluation" of expression. However, it is not defined what is the value of an expression, or what constitutes the evaluation of an expression. It would not be very difficult to supply sufficiently precise definitions.

According to clause 6.7.2.2, "The results of the real arithmetic operators and functions shall be approximations to the corresponding mathematical results. The accuracy of this approximation shall be implementation-defined." Such a specification is definitely an improvement but is insufficient. For what is an approximation? Suppose that we have a floating point system where the range of absolute values of representable numbers is roughly 1e-38 to 1e+38, and consider the operation of squaring the number 1e-30. Is 0.0 an approximation to the result? Most mathematicians would say no. And what about squaring 1e+30? Notice that what is commonly known as floating point overflow or underflow shall not be an error according to the draft. Assumably a processor may give a runtime warning; but it must also proceed using some "approximation" to the result. Notice also that clause 6.6.6.2 specifies that sqr(x) is an error if the square of x does not exist; this can be interpreted so that underflow or overflow in the calculation of sqr(x) for real x would be an error; why should sqr be exceptional in this respect?

It should be specified that the order of evaluation of the expressions of the member-designators of a set constructor is implementation-dependent. Currently no order is specified, which should probably be interpreted so that the order is implementation-dependent, but this should be stated explicitly.

2.8 STATEMENTS

The requirement (in 6.8.3.9) that "The statement of a for-statement shall not contain an assigning-reference --- to the control-variable of the for-statement." is understandable from the security point of view. However, it requires a complication of processors which would not be otherwise necessary (at least partial cross-reference information must be gathered). This means extra costs, the benefits being questionable. These comments of course only apply to checking against assigning references in procedures and functions invoked within a for-statement. One solution would be to require that the variable used as a control variable shall not be used outside that statement part in which the corresponding for-statement occurs. This would hardly decrease the expressive power of Pascal. It is moreover good programming practise to reserve the control variables for that purpose only. Such a restriction would allow the rule mentioned to be formulated in a manner which can be implemented with no significant extra costs. Notice that speaking of implementation in this context refers to inherent problems of implementing the requirement of the draft, not to any particular implementation.

2.9 INPUT AND OUTPUT

The effect of read(f,v) when f is a textfile and v is of integer or real type is incompletely specified in clause 6.9.2. It is said that it causes "reading from f a sequence of characters", and assumably reading involves the same operation as get. However, the details are unspecified. The error condition descriptions use the notion of "the rest of the sequence", but it is left undefined what "the sequence" is; a related rule ("Reading shall cease ---") is given, but it is obscure. For instance, if the characters "1", "E", and "X" are encountered, in that order, when reading a real number, what happens? Most existing runtime systems report a format error, but the specification of the draft would seem to imply that the input should be accepted, "1" being the longest sequence available that forms a signed-number. It is not only difficult to implement the lookahead required; such lookahead would be quite contrary to the fundamental ideas of file handling in Pascal.

It is said, in 6.9.2 (b), that "It shall be an error if the rest of the sequence does not form a signed-number according to the syntax of 6.1.5.". This purely syntactic approach gives no answer to the question how underflow or overflow should be treated.

The definitions (c) and (d) in clause 6.9.2 should be given by appropriate equivalent program fragments or other uniquely interpretable methods.

2.10 PROGRAMS

The note in clause 6.10 is very obscure. What are the properties of a Pascal program?

The pragmatic meaning of sample program t6p6p3p3d2revised as test program should be enlightened. Moreover, the program is related to earlier versions of draft standards (the program is not related to clause 6.6.3.3 as one would expect), and should be accordingly updated.

2.11 HARDWARE REPRESENTATION

Comment delimiters should be required to be matching, so that comment beginning with "(*" is only closed by "*)" and comment beginning with "ä" is only closed by "å". In fact, clause 6.1.8 should be rewritten in this respect, so that there would be two different forms of comments. The character "å" (as well as "ä") has been replaced by a national letter in several modifications of international character codes

2.12 TYPOGRAPHIC ERRORS AND STYLISTIC MATTERS

The table of contents does not correspond to the titles in the text (e.g. for clauses 6.1 and 6.2).

Clause 3.4 should say "accepts a program" instead of "accepts the program", i.e. accepts any program (subject to 1.2 (a)).

The specification of char-type in 6.4.2.2 would be better formulated if the beginning of the second statement would read as "The values shall be the enumeration of an implementation-defined set of characters". Similar comment applies to the pseudo-production for "string-character" in 6.1.7.

In 6.2.2.9, the word "new-pointer-types" should appear in singular, because it is preceded by "any".

In the final note in clause 6.4.3.2, the comma following the word "which" is ungrammatical. (Possibly it should precede the "which".)

In 6.4.3.5, the paragraph beginning with "Let f.L and f.R each be a single value ---" uses the word "single" redundantly in two occurrences.

In 6.4.4, the comma after the word "them" in the second statement is ungrammatical.

The abbreviated notation specified in 6.5.3.2 and 6.6.3.7 is described by saying that "a single comma" or "a single semicolon" replaces a certain syntactic construct. The word "single" in these contexts is redundant.

In 6.6.3.6 (e) (1), the word "index-type-specification" is misspelled as "index-type-specifiecation".

In 6.6.5.3, the second statement of the specification of the second form of new contains the misspelling "possesed" of "possessed".

In 6.9.4.5.1, the specification of the condition under which the sign character is '-' involves the condition (eWritten < 0). However, it seems to be so that (e<0) implies (eWritten>0) so that the latter can be omitted. Probably the redundancy results from an analogy with 6.9.4.5.2. (For fixed point representation the condition (e<0) and (eWritten>0) does not contain redundancy, of course.)

ATTACHMENT E

COMMENTS FROM THE FRENCH MEMBER BODY
ON ISO/TC 97/SC 5 N 595
SECOND DP 7185 - SPECIFICATION FOR
COMPUTER PROGRAMMING LANGUAGE PASCAL

GENERAL

The French committee voted positively about this second draft proposal, one of its main motivations being that the standardization of PASCAL will be useful only if it is completed very soon. As a further way to speed up the remaining part of the standardization process, the French member body strongly suggests that the next meeting of WG 4, whose main purpose will be to revise and incorporate if possible those improvements suggested during the vote, do not wait until the next meeting of SC 5 in London, but is convened before summer. The French member body officially offers to organize such a meeting in NICE, France, in June or July of 1981. This should allow the completion of the standard to be done in the present year.

The following comments are devided in two parts : technical comments, which deal with the language PASCAL as described in the second DP 7185, and editorial comments, which deal with the description itself. Comments considered especially important by the French member body are emphasized with an asterisk.

COMMENTS

* Character set, special symbols, reference language

The French committee tried several times, but with no success, to obtain the specification in Standard Pascal of a required character set, and to obtain a clear separation between the description of the reference language and its various hardware representations. The current state of the draft proposal shows that these proposals were not so bad, since, while the printing quality and the character set of the descriptions of Pascal are quietly worsening from one version to the next, they become at the same time more and more similar to the current ISO standard character set. The last evidence of this progressive modification is the replacement of the character "↑", the only remaining one that was not in the ISO set, by the character "^". Although these modifications result only from successive changes in the printing devices used for the successive descriptions, some benefit can be got. Hopefully, the final version of the standard description will not use a printer with only the 48 character set of Fortran !

The main concern of the French committee is that the lexical description of Pascal does not prevent the use of good printing devices with their full range of capabilities, i.e. that Pascal programs printed with boldface keyboards, italics identifiers and not-too-offending operators (for exemple, in both Wirth's books published by Printice-Hall) are legal Pascal programs.

This does not deal only with books, after all, since the time when phototypesetters or printing devides of an equivalent quality will be usable for ordinary computer output is probably not so far.

Although a clear distinction between the reference language and its hardware representations would have been considered by the French committee more appropriate for such a purpose, the current draft allows almost completely what we need, in a different way. Since the representation of letters is considered insignificant, the only remaining problem is with special symbols. Alternative representations were provided for implementations which lack some good quality characters, like square brackets or braces. In the present draft, an alternative is provided for implementations which have a better character than "^", i.e. the up arrow. We propose to prusue in such a direction, and to provide good alternatives for unsatisfying special symbols. No implementation is required to provide these alternatives if they are not available in its character set, but a program which uses them is legal. Our proposal of course, does not include bad representations for existing good symbols, made only for using available characters, like "&" for "and", for example, or worse, "#" for "≠".

Proposal : table 6, page 68

Add the following alternative symbols, which appear in the order of decreasing importance :

reference	<>	<=	>=	and	or	not
alternative	≠	≤	≥	∧	∨	¬

* Conformant array parameters

The French committee tried to compare the four successive variants of the proposal that were done in the first DP 7185, in WG 4 documents N 5 and N 9, and in the current DP. The main critic we made about the current state of the proposal is that a feature added for a very precise purpose (i.e. to allow character string constants as conformant array parameters) is now used for a completely different thing, reminiscent of PL/1 (i.e. simulating value parameters with dummy variables).
What is worse, the first intended purpose is not completely achieved, since a formal conformant array parameter cannot be a string variable, which greatly weakens the advantages provided by the feature. Several possible solutions were considered.
The proposal we made seems to have only very simple consequences on both the description of the language and its implementation, it needs no modification to the level 0 conformity, and its has interesting consequences on most uses of conformant array parameters.

Proposal 1 : Section 6.6.3.7, pages 35 to 37

Come back to the wording of WG 4 N 9, or something equivalent which uses an auxiliary variable only when the actuel parameter is a string constant, and moreover which does not force any implementation of the feature.

Proposal 2 : Sections 6.4.3.2, 6.6.3.7 and 6.6.3.8

In Section 6.6.3.7, allow the lower bound of an index-type-specification to be a constant of the suitable type, in which case the corresponding actual parameter must have an index type with the same lower bound.

Conformant array parameters with a constant lower index bound would probably be the great majority, and they can be implemented more efficiently. Moreover, in Section 6.4.3.2, extend the definition of a string type to include the case of a packed conformant array of characters with a constant lower bound of 1. Thus the formal parameter is a string, comparison operators can be used as well as the procedure write, and it should only be stated that it is an error when upper bounds differ in an assignment involving such "conformant strings".
Of course, the two preceeding proposals should be carefully worded, and all consequences on the full draft taken care of. This could be done for the next meeting of WG 4.

Recursive type definitions

On page 12, Section 6.4.1, the last sentence of the paragraph that follows the syntax makes an exception to a general rule, especially for allowing the use of a type-identifier in a pointer-type, while it is not entirely defined, as in the following example :

```
type T1 = record ... x : ↑T1 ; ... end ;
```

Of course, this is not necessary, since the type ↑T1 may be defined ans named before, an probably this definition is needed anyway for other purposes, because of the strict compatibility rules. What is worse, this exception legalizes some absurd type definitions, as in the following example :

```
type T2 = array [1..100] of ↑T2 ;
     T3 = ↑T3 ;
```

Proposal : Section 6.4.1, page 12

Remove the first half of the last sentence of the second paragraph, which thus becomes :

"The type-denoter shall not contain an applied occurrence of the identifier in the type-definition".

The required type integer

On page 48, Section 6.7.2.2, the first paragraph implies that there may exist some values of the integer-type that are not in the closed interval -maxint..+maxint. This seems useless. On the contrary, on machines using two's-complement arithmetic, the negative number with the largest absolute value could be used as an "undefined" value, extremely useful for checking that variables are initialized.

Proposal : Section 6.7.2.2, page 48

Reword the first paragraph so that the integer-type is exactly the interval -maxint.. +maxint.

EDITORIAL COMMENTS

- page 2, 1.2 (a)

  Add the sentence ", and the actions to be taken when the corresponding limits are exceeded".

  This suggestion was triggered by the constatation that nothing was said about what happens when the procedure new finds no more available space.

- page 7, 6.1.5

  Nothing is said about the meaning of the period and the digit-sequence that follows it, in an unsigned-real. A possible solution would be to replace "digit-sequence" with "fractional-part", defined elsewhere as a digit-sequence.

- page 10, 6.2.3

  This whole section is very difficult to understand. A possible solution would be to use a simple stack implementation model, not compelling for implementators, but much clearer.

- page 11, 6.3

  This is the first occurrence of a systematic principle used in the whole standard, i.e. identifiers are always quelified in syntax rules, except for their defining-point. This is pretty good, but a note should explain it, for example, at the end of Section 6.2, or in Section 4.

- pages 15, 18, 19

  Examples use type identifiers that are defined only on page 22 (colour, vector) or not defined at all (string, angle). Something would be done.

- pages 33, 34

  Boring repetitions occur every time something is saif about procedures and functions. By defining the term "subprogram", and by specifying a uniform substitution with either "procedure" or "function", it should be easy to simplify and shorten the second paragraph of page 33, the last two paragraphs of the same page, and Sections 6.6.3.4 and 6.6.3.5 on page 34.

- page 34, 6.6.3.3

  Since the types possessed by the actual-parameters are the same as that denoted by the type-identifier, they must be identical. The second sentence of Section 6.6.3.3 is consequently useless.

- page 35, 6.6.3.6

  By replacing in (a) the two occurrences of "value" with "value(resp. variable)", it is possible to entirely omit (b).

- page 36, 6.6.3.7

  A note should be insered before the last paragraph of page 36, explaining that bound-identifiers are neither constants nor variables.

- page 37, 6.6.3.7

The first sentence of the second paragraph is impossible to understand, and probably wrong. The fourth paragraph is extremely difficult to understand, and should be either worded differently or illustrated with an example, or both. In the third note of the page, "anonymous" should be replaced with "auxiliary", for uniformity.

- page 43, 6.6.6.4

The descriptions of succ and pred differ only by one word ("less" instead of "greater"). A simplification in the same way as page 35, 6.6.3.6 should be possible.

- page 47, 6.7.2.2

The last three paragraphs of the page begin with a sentence stating that a term is an error if something occurs. Given the definition of an error, it should be better to state that it is an error if y = 0 in a term of the form x/y, etc.

- page 50, 6.7.3

For the sake of uniformity with Section 6.8.2.3, the second sentence should end with "... activation of the block of the function-block associated with the function-identifier of the function-designator".

- page 52, 6.8.2.4

The wording is extremely unclear, especially in (b). What are "these exceptions" ?

- page 53, 6.8.3.5

By adding ", otherwise it shall be an error" at the end of the first paragraph, the second one can be omitted.

- page 55, 6.8.3.9

Nothing is said about the assignment-compatibility of the initial-value.

- page 59, 6.9.1

It seems that only textfiles occurring as program-parameters could be used at all. This relates to nothing elsewhere, and should be omitted.

- page 68, 6.1.1

The last part of note 2, dealing with the possibility of national variants, disappeared during the summer. Why ?

- page 67

The chosen example cannot be considered a significant demonstration of the capabilities of Pascal. A better example could be found in one of the numerous textbooks about the language.

- Appendices

Syntax diagrams are recognized as an excellent means for syntactic descriptions, especially for Pascal. They should be included in an additional appendix.

ATTACHMENT F

Part I. Technical reasons

1. Call-by value for conformant array parameters

We do not approve that the call-by-value of conformant array parameters is specified by enclosing the a c t u a l parameters in parentheses. In Pascal, the parameter access method is always specified with the f o r m a l parameters. There should be no exception for conformant array parameters.

2. Use of "denote"

The use of "denote" in Second DP 7185 is not consistent. See the accompanying notes "German concerns on the use of 'denote'".

Part II. Editorial comments

0. INTRODUCTION
Delete this heading and include the text as new paragraph 1.3 .

4. DEFINITIONAL CONVENTIONS, Table 1
Delete the line "> shall have as an alternative definition".

5.1 Processors (h) and (i)
Replace "specified for errors" by "specified for violations".

6.1.5. Numbers
Change the sequence of the syntax to run from signed-number to digit-sequence (top-down) in accordance with usage in other places of the Second DP 7185.

6.2.3.2 (d) and (e)
Formal parameters are associated to the b l o c k , not to the identifier (see 6.6.1). Change, therefore, the wording as foolows:

(d) for each procedure-identifier local to the block, a procedure with the procedure-block corresponding to the procedure-identifier, and the formal parameters of that procedure-block; and

(e) for each function-identifier local to the block, a function with the function-block corresponding to, and the type possessed by, the function-identifier, and the formal parameters of that function-block.

6.4.2.2 integer-type

Include after "see also 6.7.2)." the following text taken from 6.7.2.2: "The required constant-identifier maxint shall denote an implementation-defined value of integer-type. All integral values in the closed interval from -maxint to +maxint shall be values of the integer-type."

6.4.1 General. Second paragraph.

Replace "as the domain-type" by "in the domain-type".

6.4.1 General. Third paragraph.

Delete the sentece "The required types shall be denoted by predefined type-identifiers (see 6.4.2.2 and 6.4.3.5)."

6.4.2.2 char-type

Insert after "without graphic representations" the following text ", the others denoted as specified in 6.1.7 by the character-denoter".

6.4.2.3 Enumerated types.

Delete "as their identifiers occur ... enumerated-type" and add after "from zero." the following: "The mapping shall be order preserving."

6.4.3.1 General.

Change the sequence of the syntax to run from new-structured-type to structured-type (top-down).

6.4.3.2 Array-types. Next to last paragraph.

Insert after "a smallest value of 1" the following: "and a largest value of greater than 1". This is a clarification for the use of string types.

6.4.3.2 Array-types. Last note.

Delete comma after "which".

6.4.3.4 Set-types.

Replace "of its base-type" in the first sentence by "of the base-type of the set-type".

Replace "an unpacked set designated" in the last paragraph by "an unpacked set type designated".

6.4.3.5 File-types. Last four paragraphs.

Replace "a sequence x~S(e), where x is" by "a sequence cs~S(e), where cs is".

Replace "If x is a line then no component of x other than x.last" by "If l is a line, then no component of l other than l.last".

Replace "A line-sequence, z, shall be either the empty sequence or the sequence x~y where x is a line and y is a line-sequence" by "A line-sequence ls shall be either the empty sequence or the sequence l~ls' where l is a line and ls' is a line-sequence".

Replace in (b) the text "shall be x~y where x is a line-sequence and y is a sequence of components" by "shall be ls^cs where ls is a line-sequence and cs is a sequence of components".

In the NOTE following (b) replace y by cs in two places.

6.4.7 Example
In NOTES 2. replace "to have been declared" by "to have been defined".

6.6.1 Procedure-declarations. Third paragraph.

Replace "the the procedure-declaration" by "the procedure-declaration".

6.6.3.6 Parameter list congruity.
In (e) (1) replace "index-type-specifiecation" by "index-type-specification".

6.6.3.7 Conformant array parameters.

We propose to use the syntax as stated in "Notes on US concerns".

6.6.5.2 File handling procedures. First paragraph.

Move the clause "and similarly for f0^ and f^" to the end of the sentence.

6.6.5.3 Dynamic allocation procedures. NOTE.
Replace "see 6.8.2.2" by "see 6.8.2.2 and 6.6.3.2" .

6.7.2.2 Arithmetic operators.

The paragraph after the NOTE shall read as follows:
"Any monadic operation performed on an integer value in the interval -maxint..+maxint shall be correctly performed according to hte mathematical rules for integer arithmetic. Any dyadic integer operation on two integer values in this same interval shall be correctly performed according to the mathematical rules for integer arithmetic, provided that the result is also in this interval. Any relational operation on two integer values in this same interval shall be correctly performed according to the mathematical rules for integer arithmetic."
(Note that the other parts of this paragraph have been shifted to 6.4.2.2.)

6.7.2.4 Set operators. Table 4.

Insert after "a canonical set-of-T type" the following: "(see 6.7.1)".

6.7.2.5 Relational operators. Table 5.

Delete "(see 6.7.1)" after "a canonical set-of-T type".

In the fourth paragraph after Table 5, replace "Where u and v denote simple-expressions" by "Where u and v denote operands".

6.8.1 General.

Replace "A label occurring in a statement" by "A label, if any, of a statement".

6.8.2.2 Assignment-statements.

Delete the last paragraph "The state of a variable ... possess a structured-type." Insert this text under 3. DEFINITIONS as 3.5 undefined. and 3.6 totally-undefined.

6.8.2.3 Procedure-statements. First paragraph.
In the text "which is list of" insert an "a" after "which is".

6.8.3.5 Case-statements.

Delete last sentence of the first paragraph "One of the ... to the case-statement."

6.8.3.9 For-statement.

Replace "The value of the final-value shall be assignment-com= patible with the control-variable" by "The value of the final-value shall be assignment-compatible with the type possessed by the control-variable".

6.8.3.10 With-statements.

Replace "as the only record-variable" by "as single record-variable".

In the Example replace "shall be equivalent to" by "shall "has the same effect on the variable date as" .

6.9.2 The procedure read.

(c) Delete the clause "the longest sequence available that forms". Change the sequence of the last sentences.

(d) same as section (c).

6.9.4.1 Multiple parameters.

Delete the heading; preserve the text as part of 6.9.4.

6.9.4.2 Write-parameters.

Change to 6.9.4.1.

6.10 Programs. First paragraph.

Replace "Each program parameter shall be declared" by "Each program parameter except the identifiers input and output, if occurring, shall be declared".

Second example: Replace "t6p6p3p3d2revised" by "t6p6p3p4d2revised" .

German concerns on the use of "denote"

In the use of the word 'denote', we realize the insight that there exists a sharp difference between the 'thing' meant by a certain piece of program text, and the program text itself. All kinds of syntactic constructs never are those mysterious P a s c a l things, but only denote them.

NOTE: This distinction may be found in some formal language definition techniques, especially the denotational semantics (see Gordon, Stoy, Tennent, Bjorner/Jones).

We fully agree with an approach allowing us to treat the P a s c a l objects without need to refer to some syntactic instances, and we feel it the only way to succeed in drafting an unambiguous and yet understandable standard.

Unfortunately, however, the promising approach has not been carried throught the whole draft, what lack, on the one hand, makes it even more ambiguous than former, not formally based, drafts, and on the other hand, at some points totally unclear.

As an example for the latter conjecture look at 6.6.3.7 of N9. There is stated on p. 18, line 8f: "...the formal parameters shall possess an array-type ...", and in the NOTE on the same page: "The type of the formal parameter cannot be a string-type (see 6.4.3.2) because it is not denoted by an array-type."

For the initiated, the word "denoted" in the note makes clear that the latter "array-type" means a piece of text derivable from the syntactic non-terminal array-type (p.15 of N4), while the former means a semantic entity, a property of a variable structured as an array. Is every reader of the standard initiated?

The following lines list those places in N4/N9, where we found errors in the two drafts related to the "denote"-distinction between syntactical and semantical entities. We do not claim for completeness!

- 6.4.2.1: Simple Types General: we are not able to derive the real-type (integer-type, boolean-type,char-type) from simple-type, but only the denoting identifiers.
  simple-type = ordinal-type I real-type-identifier
  ordinal-type = new-ordinal-type I integer-type-identifier I Boolean-type-identifier I char-type-identifier

- 6.4.3.2 Array-types: the second to sixth occurence of the word "array-type" in the section address the synctactic entity , the others the semantical thing, the mapping.

NOTE : We assume that all sections on type specify the same mess, but do not list all of them.

- 6.4.3.4 Set-types: In the last paragraph "S" seems to be the name of the semantical thing, but the wording "set of S" instead of set-of-S supports the syntactical view. In either case, it is used wrongly.

- 6.5.1 Variable-declarations: In the second paragraph, "buffer-variable" is used for both, the syntactical structure and the semantical entity.

- 6.6.3 Parameters: Formal parameters and actual-parameters are syntactical entities and do not possess a type! The type is possessed by the variable denoted by the parameters.
  Here we have a real clash in terminology, because we should better associate the type of a formal variable parameter with the parameter-identifier, not with the denoted variable, since the denoted variable is the variable denoted by the corresponding actual-parameter.

- 6.6.5.2 File handling procedures: On p. 38 the verbs "to denote" and "to be" are used just the false way round. Some examples: "vl...vn denote variable-access" should read "vl...vn are variable-accesses", "Consequently it may be a component of a packed structure" should read "Consequently it may denote a component of a packed structure", since variable-accesses are pieces of text (like vl) denoting variables (like components of packed structures).
  Additionally, only the variable denoted by the file-variable f possesses a type, and read, readln, write, writeln are not procedures, but procedure-identifiers.

- 6.6.5.3 Dynamic allocation procedures: P is a variable-access (a statement missing in the draft!) and denotes a variable, which possesses a type and may be attributed a value.

- 6.6.5.4 Transfer procedures; A can be a variable-access, not a variable, j and k don't possess types, and an expression does not have a value.

N O T E : It is impossible to list all inconsistencies of 6.6.4, 6.6.5 and 6.6.6. We assume that these section have not been untergone careful reading when introducing the distinction between syntax and semantics.

- 6.7.1 Expressions General: The first sentence states, how it should be: "An expression shall denote a value". The last paragraph on p.43 and the NOTE, however, miss a number of "denote"s: "shall have the value denoted by x", "from the value denoted by x to the value denoted by y", "if the value denoted by x greater than the value denoted by y".

- 6.7.2.5 Relational operators,

- 6.7.3 Function designators,

- 6.8.3.4 If-statements, and

- 6.8.3.7 Repeat-statements: Here we find the word "yields", which (possibly) reflects the fact, that the values denoted by the expressions are time-variant. We will comme to this point later.

- 6.9 Input and Output: The points of 6.6.5.2 as to "to be", "to denote", "to possess a type" and to the distinction between procedures and procedure-identifiers apply here, too.

As we have tried to show, the introduction of the syntax/semantic-distinction, which made the draft much harder to read than its predecessors, resulted, as undergone only half-hearted, in a draft being neither exact nor readable, while former ones were at least readable.

We do not think that correction of all errors (or laxities) will do, as the standard, then, will be totally unreadable. Instead, we have two alternative proposals for further processing:

1) Pull the approach to its end, but in a more suitable form, i.e. give a formal definition of PASCAL based on Oxford notation or the related and more convenient Vienna Development Method. This will establish an unambiguous reference for implementors and debuggers. Additionally, for the informal reader (he who would have been content with one of the former drafts) annotate the formal definition with some text along the lines of one of the former drafts.

2) Make the distinction between syntax and semantic totally clear by consequent wording, e.g. a syntactical non-terminal denoting some semantical entity x should be specified an "x-denoter". Pushing this approach through the draft will at least convert all inconsistencies and ambiguities into errors, which may be fixed by two ways, an exact one and a lax one:
The exact one proceeds by inserting the words "denoted by" at all places where they are needed. As we mentioned earlier, the draft will probably become unreadable. The lax one includes the sentence: "Wherever context makes clear whether an x or an x-denoter is addressed, the x-denoter is used to name the x". Then we may throw away a lot of "denote"s and have to correct only some places (e.g. the first mentioned section on conformal-array parameters).

NOTE: We like proposal 1 better, since it is more clean and thus more suited for an international standard.

At last, a few words on the deferred time-variance problem: The relation between a variable-access or a function-designator and its value is not as simple as the relation between a type-denoter and its type, but is twofold: the variable-access denotes a variable, and that variable "denotes" the value actually attributed to it. The semantics of an assignment statement is a change only of the second relation, while a procedure call affects the first one. So we should not use the word denote to describe the relation between a variable-access and its value, and, as expressions incorporate variable-access, an expression and its value.
In the denotational semantics the two-stageness is reflected by the use of two different mappings, one relating the synctactical to the semantical entity, and one relating that to the value: By this, you can clearly describe how different operations (assignment versus call) affect different changes in meaning.

References:
Bjorner D., Jones C.B (eds): The Vienna Development Method: The Meta-Language, LNCS 61, Springer 1978
Gordon M.J.C.: The Denotational Description of Programming Languages, Springer 1979
Stoy J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT Press 1977
Tennent R.D.: The Denotational Semantics of Programing Languages, CACM 19 (1976), 8, 437 - 453

ATTACHMENT G

Japanese Comments

We saw that the second draft proposal (N 595) had been extremely improved. The elaboration done by the editors shall be highly appreciated. However, the proposal still contains several problems to be considered carefully and, because some of them are very essential, we are very sorry to disapprove the draft this time once again. Our comments are as follows.

1. Scope rules (6.2.2)

1.1 According to 6.2.2.4, the rules 6.2.2.5 and 6.2.2.6 shall be exclusion principles. From this viewpoint, rule 6.2.2.5 seems all right. However, 6.2.2.6 shall be amended as:
6.2.2.6 The region that is the field-specifier of a field-designator shall be excluded from the enclosing scopes.
The original 6.2.2.6 expresses the same rule as one expressed in 6.5.5.3 and thus seems superfluous.

1.2 6.2.2.7 shall be amended as:
6.2.2.7 There shall not be two defining-points of the same identifier or label for the same region. The original 6.2.2.7 'The scope of a defining point of an identifier shall include no other defining point of the same identifier' does not allow, say, the occurence of the value parameter identifier because (see p.33) the scope that is the formal parameter list of the defining point as a parameter identifier contains the defining point as the associated variable identifier for the region that is the block.

2. Conformant array parameters

2.1 We have discussed on this matter very intensively and came to conclude that the conformant-array-parameters in the present form is still too ad hoc and premature. It makes it very hard to teach or explain the language. It contradicts with the original aim of the language that is 'to make available a language suitable for teaching programming as a systematic discipline based on certain fundamental concepts clearly and naturally reflected by the language'. If the conformant array parameters shall be introduced for 'writing of both system and application software', the inclusion of only conformant array parameters seems not enough. We need more features. So, we strongly recommend to remove the conformant array parameters from the current draft. It shall be reconsidered together with other important extensions, after the current draft is standardized.

2.2 Especially we don't like the feature to indicate value and variable parameters at the calling site. This is not the principle of Pascal but of Fortran. We can not accept the mixture of Pascal and Fortran.

2.3 Descriptions for the conformant array parameters have not been brushed up. The sentence like 'The actual parameter shall be either a variable access or an expression that is not a factor that is is not a variable access' is beyond our understanding. Moreover, in the same clause, there are several places where the expressions are meant in this sense without any comments. We think it would take long to improve the idea of the conformant array parameters. So, in order to approve the draft in one or two more editings, the discussion of the conformant array parameters shall be postponed to the later version.

3, Syntax rules

3,1 Groups of syntax rules in a clause are presented bottom-up (cf, expression 6,7,1) or top-down (cf, record types 6,4,3,3) or in mixed order (cf, structured type 6,4,3,1). They shall be presented in a systematic way.

3,2 Throughout the whole syntax rules, there are nonterminal symbols which are defined but not referred to in other rules. They are only used in semantics. They are: pointer-type, program, read-parameter-list, readln-parameter-list, special-symbol, signed-number, simple-type, structured-type, write-parameter-list and writeln-parameter-list. They shall be indicated as such. (For instance with an asterisk as in ALGOL 68,) There are nonterminal symbols that are referred to but not defined. They are: field-designator-identifier, integer-type, boolean-type, char-type and real-type. field-designator-identifier shall be defined. Others shall be indicated.

4, 8 character rule for identifiers and 4 digit rule for labels
If the eight character rule is not adopted then the four digit rule shall be removed.
6,1,6 'that shall be in the closed internal 0 to 9999' -> empty.

5, Sequence type rules 6,4,3,5
In rule (c), component c is also concatenated from the right to define ,last like x^S(c). So, the rule (b) shall be amended: 'and S(c)^x and x^S(c) shall also be a sequence.' As a whole, the preciseness of description of the draft varies excessively from place to place. Accordingly the draft makes readers find the composition very unbalanced . We believe English speaking people will naturally feel the points by far more sensibly than we did.

6, New-type
Types are denoted either by type-identifier or new-type. See p,12,
type-denoter = type-identifier | new-type .
ordinal-type = new-ordinal-type | ... |
ordinal-type-identifier .
So, similarly array type shall be
array-type = new-array-type | array-type-identifier .
The type-identifier vector shall be the array-type-identifier, not the structured-type-identifier. And so on.

7, Editorial comments
p,7 unsigned-real = unsigned-integer('.' ... |'e' ...) .
p,9 1,-14 Add 6.10 (defining point for input and output)
p,22 1,17 (a) T1 and T2 are the same type which is permissible as a component type of a file type. (This is not the only place where rules are to be interpreted recursively. Remark for recursiveness shall be treated evenly.)
p,28 1,23 the the -> the
p,28 1,26 'forward' -> forward (In 6,1,4 forward is used without quotes.)
p,29 Insert '(* This example is not for level 0,*)' to procedure declaration AddVectors,
p,31 1,4 the the -> the
p,31 1,8 'forward' -> forward
p,31 1,15 Example of a procedure-and-function-declaration-part -> Example of a procedure-and-function-declaration-part ;
p,36 1,7,8 (packed-conformant-array-schema | unpacked-conformant-array-schema) -> packed-conformant-array-schema | unpacked-conformant-array-schema
p,36 1,23 contains -> closest-contains
p,36 1,25,26 ']' 'of' 'array' '[' ->] of array [(Word symbols are not quoted outside the syntax rules,)
p,38 1,13,14 is is -> is
p,40 1,11 Insert 'write' and adjust indentation,
p,40 new(p)! Indicate that p is the variable parameter,
p,41 pack(a,i,z)! Indicate that z is the varible parameter. And so on,
p,48 1,1 Add 'and J > 0' after 'i >= 0',
p,51 1,-20 or to the function-identifier -> or to the function denoted by the function-identifier (see 1,-9 when the variable or function does not have attributed ,,,)
p,52 Insert '(*This example is not for level 0,*)' to procedure statement AddVectors,
p,53 1,2 6,8,3,3 conditional-statements, -> 6,8,3,3 conditional-statements (remove period, see 6,8,3,4 if-statements)
p,53 1,-6 Delete 'one of the case-constants ... to the case-statements,' because the same meaning is containded in the next sentence 'it shall be an error if ... upon entry to the case-statement.'

USA Comments on 97/5 N 595 - 2nd Draft Proposal 7185 - Pascal — ATTACHMENT H
Comment on Section 6.6.5.3 — PART I

Status: Error

PROBLEM:

The current draft (7185/2) says it is an error to provide Dispose with fewer tag arguments than were given New to create the object. The requirement that m not be less than n is to avoid disposing more space than was originally allocated. However if m is greater than n, then it is approved to dispose less than was originally allocated and leave a dangling piece of storage space that cannot be reclaimed. It should be an error if the tag field list in dispose is not identical to its corresponding new. The argument that this may be too hard to detect is vacuous because, in the form "it shall be an error...", its detection is optional.

RECOMMENDATION:

Change "m is less than n" to "m is not equal to n".

Comment regarding functions

STATUS: Error.

PROBLEM:

DP7185/second edition does not currently specify function results. In particular, assignment to a function-identifier has the effect of attributing a value to the function instead of to an activation of the function. This ignores the problem of functions for which there exist more than one activation.

Thus, for example, the following program will write the sequence of integers (2,1,0) according to the commonly held interpretation, but will write the sequence (2,2,2) according to the specifications in DP7185/second edition.

```
program p(o); {a "counter" example}
  type natural = 0..maxint;
  var  o: file of natural;
       count: natural;
  function f: natural;
  begin
    f := count;
    if count <2 then
      begin count := count + 1; write(o,f) end
  end;
begin rewrite(o); count := 0; write(o,f) end.
```

The solution to this problem requires the introduction of a new part of an activation of a function which has many of the characteristics of a variable. This is a nontrivial change and requires alterations to 6.2.1, 6.2.3.2, 6.2.3.3, 6.6.2, 6.7.3, and 6.8.2.2.

PROPOSED CHANGES:

In 6.2.1, last sentence, insert after the second comma:
and any result of an activation.

In 6.2.3.2, replace (e) with:
(e) for each function-identifier local to the block, a function with the formal parameters associated with, the function-block corresponding to, and the result type associated with the function-identifier; and
(f) if the block be a function-block, a result possessing the associated result type.

In 6.2.3.3, paragraph 2, append the clause:
; except that the function-identifier of an assignment-statement shall, within an activation of the function denoted by that function-identifier, denote the result of that activation.

In 6.6.2, paragraph 3, change "possessing the type denoted" to:
associated with the result type denoted

In 6.6.2, paragraph 2, replace sentence 2 with:
A function-block shall contain at least one assignment-statement such that the function-identifier of the assignment-statement is associated with the function-block.

In 6.6.2, paragraph 2, delete the last 2 sentences (revised restrictions are incorporated into 6.7.3, which is where they always should have been.)

In 6.6.2, append the following the paragraph 5:
; the block of the function-block shall be associated with the result type that is associated with the identifier or function-identifier, respectively.

In 6.7.3, paragraph 1, replace sentences 1 and 2 with:
A function-designator shall specify the activation of the function denoted by the function-identifier of the function-designator, and shall yield the value of the result of the activation upon completion of the algorithm of the activation; it shall be an error if the result is undefined upon completion of the algorithm.

In 6.8.2.2, paragraph 1, replace sentence 1 with:
An assignment-statement shall attribute the value of the expression of the assignment-statement either to the variable denoted by the variable-access of the assignment-statement, or to the activation result that is denoted by the function-identifier of the assignment-statement; the value shall be assignment-compatible with the type possessed, respectively, by the variable or by the activation result.

In 6.8.2.2, paragraph 3, sentence 1, change "variable or function" to "variable or activation result" (twice), and in sentence2 and 3 change "variable" to variable or activation result" (4 times).

JUSTIFICATION:

Corrects an error.

Comment on document X3J9/81-007

(Dr. Arthur Sale's letter to Dr. Addyman of January 12, 1981.

Status: Change

Observation:

We have reviewed the document cited above. We took particular note of items AHJS-81/5 "definition of error" and AHJS-81/6 "definition of processor".

We concur with Dr. Sale's evaluation and recommendations regarding these items..

Comment on 6.9.1 I/O (page 59)

Status: Editorial

Problem:

The term "legible" is not well defined and the whole paragraph is unnecessary.

Proposed change: Delete clause 6.9.1.

Comment on 5.1 Processor Compliance

Status: Change

Problem: Clause (e) doesn't really require anything.

Proposed Change: In clause (e), replace "detect" with "detect and report".

Justification:

The change to clause (e) requires the processor to diagnose violations of the standard, at least at user option.

Comment on 6.2.1 Blocks

Status: Editorial

Problem: The first and last paragraphs of this section are not about blocks and should be elsewhere in the text.

Proposed Change:

A new sub clause between 6.2 and 6.3, should be created, and titled "Labels". The first paragraph of 6.2.1 should become the text of this sub clause.

The last paragraph of 6.2.1 should become the first paragraph of 6.2.3.5.

Justification:

Each of the other declaration parts of the block has a section to itself, viz.: 6.3 constants, 6.4 types, 6.5 variables, 6.6 procedures. For parallelism, and so that the user may be able to find it, labels should have a parallel section, however small.

The last paragraph of 6.2.1 is one of the activation rules and belongs next to the rule on the life of variables in 6.2.3.5. This change also serves to organize the standard so that things may be found.

Comment on 6.4.3.5 Textfiles

Status: Error

Problem:

On page 21, the disclaimer on textfile structure does not go far enough. There is a real danger that some officially sanctioned validation suite may contain tests such as the attached program (reprinted from JPC/80-061).

Proposed Change:

On page 21, first paragraph, replace the last sentence "This definition... processor" with:
"These provisions describe the functionality only, and shall not be construed to determine in any way the underlaying representation of textfiles; in particular, the relationship, if any, between end-of-line and values of the char-type shall be implementation-dependent."

Justification:

There is too much myth about textfiles to permit the standard to gloss over many machine dependencies with a disclaimer on end-of-line. It suggests that one doesn't expect the end-of-line to be a space and that an implementor is not required to have a character (byte) which is the end-of-line. But it does not make clear that the attached program is implementation-dependent.

Moreover, the original description in the UM&R: "text = file of char" has led to more than one implementation-dependent program which the author believed to conform to all reasonable portability considerations in the UM&R. It is therefore necessary to dispel that notion in the standard by expressly stating the implementation-dependency of textfile I/O.

```
 program testeol (output, textf);
{
       This program tests whether textfiles handle the character set
       and end-of-line interrelations properly
}
 const
       maxchr = 127    (the maximum ordinal value of type char
                        in this case the value is 127 for ASCII);
 var
       textf:  text;
       fvalue: char;
       c:      integer;
       allok:  boolean;
 begin
{      this section writes all of the char values to a textfile      }
       rewrite(textf);
       for c:=0 to maxchr do
              write (textf, chr(c));
       writeln(textf);
```

```
{
       This section reads all of the char values back
       and checks that they match what was written
}
       reset(textf);
       allok:=true;
       for c:=0 to maxchr do begin
              if eoln(textf) then begin
                     writeln(output,
                      'eoln unexpectedly returned true for c=', c:4);
                     allok:=false
              end (if);
              read (textf, fvalue);
              if fvalue <> chr(c) then begin
                     writeln(output,
                      'file value was different for chr of', c:4,
                      ' value returned was', ord(fvalue):4);
                     allok:=false
              end (if)
       end (for c);
{      this section tests for end-of-line and end-of-file    }
       if not eoln(textf) then begin
              writeln(output,
                'eoln did not return true after the last value');
              allok:=false
       end (if);
       read(textf,fvalue);
       if fvalue <> ' ' then begin
              writeln(output,
                'end of line value was not space.  It was chr of',
                ord(fvalue):4);
              allok:=false
       end (if);
       if not eof(textf) then begin
              writeln(output, 'eof did not return true at end of file');
              allok:=false
       end (if);
       if allok then writeln (output, 'textfile behaved as expectd');
       write(output, '*** end of test ***');
end.
```

Comment on various sections of the Second Draft Proposal for Pascal

Status: Editorial

Problem Statement:

There are several places where the draft proposal would be improved or corrected by minor changes in spelling, wording and punctuation.

Proposed Changes to the Draft Proposal:

p. 3: In the first paragraph of section 4 change "the identifier of a predeclared or predefined entity" to "the identifier of a required entity".

p. 11: In the last paragraph of section 6.3 change "The constant shall not contain" to "The constant in a constant-definition shall not contain".

p. 15: In section 6.4.3.2, in the paragraph that follows Example 2, change "by the index type. Then the values" to "by the index type; then the values".

p. 16: In the last NOTE of section 6.4.3.2 change "which, allow" to "which allow".

p. 19: In the paragraph following the second note of section 6.4.3.4 change "unpacked set designated the" to "unpacked set type designated the".

p. 57: In section 6.8.3.10 add the syntax definition:
field-designator-identifier = identifier.

p. 35: In section 6.6.3.6 subparagraph (e) correct misspelling of "index-type-specification".

p. 37: In the third note of section 6.6.3.7 (first note at top of page) change "can not" to cannot".

p. 36: 2nd paragraph from the bottom, replace "the first bound-identifier" by "applied occurrences of the first identifier" and replace "the second bound-identifier" by "applied occurrences of the second identifier".

p. 52: In the first paragraph of section 6.8.2.3 change "which is list of actual-parameters" to "which is the list of actual-parameters".

In 6.4.1, paragraph 2, the phrase "its type-denoter" is poor; change to "the type-denoter of the type-definition".

In 6.6.1, delete the first paragraph; the first sentence is meaningless, the second is redundant (see 6.2.3.3).

In 6.6.1, paragraph 3, change "the the" to "the".

In 6.6.1, clarify the meaning of paragraph 4 by changing "in the same procedure-and-function-delaration-part" to "closest-contained by the procedure-and-function-declaration-part closest-containing the procedure-heading".

In 6.6.1, paragraph 5, change "associates" to "shall associate".

In 6.6.2, delete paragraph 1; the first sentence is meaningless, the second is redundant (see 6.2.3.3).

In 6.6.2, paragraph 3, change "the the" to "the".

In 6.6.2, clarify the meaning of paragraph 4 by changing "in the same procedure-and function-declaration-part" to "closest-contained by the procedure-and-function-part closest-containing the function-heading".

In 6.6.2, paragraph 5, change "associates" to "shall associate".

Comment on 4. DEFINITIONAL CONVENTIONS

Status: Error

Problem: Definition of "a y containing an x" defines a y to be an x.

Proposed Change:

Reword definition to read "a y containing an x: refers to any y from which an x is directly or indirectly derived."

Justification:

The proposed wording defines a y to be a y.

Comment on ISO 2nd DP

Status: Editorial

Problem:

In previous drafts, appearances of a word-symbol or required identifier in the text were underlined when necessary to distinguish them from English words. This underlines have all disappeared in the second DP.

Proposed Change:

Restore the underlines as in previous drafts or use a different typeface. The locations affected include:

6.1.4	forward, external
6.4.2.2	integer, real, Boolean, false, true, char
6.4.3.1	packed
6.4.3.5	text
6.4.4	nil
6.6.5.2	read, write
6.7.1	not
6.7.2.2	maxint
6.7.2.5	in
6.8.3.4	then, else

Justification:

Readability is enhanced by distinguishing language elements from English words. In many of these cases, the sentence is gramatically incorrect unless this distinction is made.

Comment on Note in 6.1.4

Problem:

In 6.1.4, the note cannot be deduced from the text of the standard and is irrelevant.

Status: Editorial

Recommendation: Delete the note in 6.1.4.

Comment on 6.6.5.3 (Dynamic allocation procedures)

Status: Error

Problem Statement:

The description of the second form of dispose uses the construct "q^" where q represents a pointer expression. This use of "q^" is not defined by the draft proposed Pascal standard because an identified-variable can only be constructed from a pointer-variable and q^ is a pointer expression.

Proposed Change to the Draft Proposal: Change "q^" in the description of the second form of dispose to "the pointer value of q".

Comment on 6.10 (Programs)

Status: Error

Problem Statement:

The draft proposal requires that if the required variables input or output are specified as program-parameters then these identifiers must be declared in the variable-declaration-part of the program block. This is a change from the Pascal User Manual and Report which states that the program parameters input and output must not be declared as variables in the program block.

Proposed Change to the Draft Proposal:

In the first paragraph of section 6.10 change "each program parameter shall be declared" to "each program parameter shall have a defining-point as a variable-identifier for the region that is the program-block".

Comment on 6.8.3.5 Case-Statements

Status: Error

Problem:

The last sentence of the first paragraph is contradicted by the second paragraph. The former states the requirement that one of the case-constants shall be equal to the value of the case-index, making detection of violation mandatory (by 5.1), while the latter states the violation shall be an error, making the detection optional (by 3.1).

Proposed Change: Delete the second paragraph.

Justification:

As they stand, the two statements are obviously contradictory. The selection of mandatory detection is dictated by consistency with the majority of current Pascal implementations, rigor, robustness, and the desire to be able to prove programs correct.

Comment on Scope of procedure and function header(s)

Status: Change

Problem Statement:

The scope of identifiers appearing in procedure and function headers is unnecessarily complicated by the separation into two regions (and two scopes). This allows programs which appear contradictory, and complicates an accurate description in reference manuals. It appears to have no compensating advantages.

Example:

```
function Func(Param : integer) : integer;
type
  Integer = char;
begin
  /body of func/
end;
```

In the example, the appearances of 'integer' in the function header do not correspond to the type 'Integer' declared within the function. Specifically, type identifiers (and the procedure/function identifier) may be redefined within the procedure/function; parameter identifiers may not be redefined.

Recommendation:

Modify the scope rules so that any identifier that appears in a procedure/function (including the header) may have only one meaning throughout that procedure/function.

A possible (and desirable) effect of this change would be to prohibit redeclaration of a procedure identifier immediately within the original procedure. (Note that this redeclaration is already prohibited for function identifiers, as no assignment to the function value could be made.) Note also that this would restore the correctness of statements in sections 10 and 11 of the Revised Report: "The use of the [procedure/function] identifier ... within its declaration implies recursive execution."

Comment on 6.6.3.7 Conformant Array Parameters

Status: Change

Problem:

The technique newly introduced in dp7185 of requiring the calling procedure to determine whether a given actual parameter is to be passed by "reference" or "value" has several problems:

(1) It assigns a new semantic meaning to a syntax which formerly had a different semantic meaning - it makes the parens significant in (A).
(2) It is unlike any similar construct in the Pascal language defined by the standard
(3) This very departure from the rest of the language creates confusion for the user and leads easily to invalid programs.
(4) It creates an unnecessary limitation on implementations.

Moreover, this problem is merely the latest in a long string of difficulties in getting a technically robust conformant-array-proposal. It is not clear that it is the last such problem, since several difficulties with the previous proposals remain unsolved in the current proposal.

These problems arise out of the attempt to put the conformant-array-extension into the standard and, in particular, to do so in a strange fashion so that minimal impact on existing implementations may be felt. This approach has real penalties. We suggest four alternatives below, the first one being our preference:

(1) Remove the conformant array feature entirely and leave only the level 0 language.
(2) Allow both "value" and "var" conformant array parameters, without unusual restrictions, in exactly the same way that "value" and "var" parameters of any other type may be specified, admitting that this may require runtime specification of the size of the activation record in some instances; or
(3) Delete the "value" conformant-array-parameter construct entirely, and therewith the attempt to permit string manipulation via conformant array parameters.
(4) Consider as an alternative for further study document JPC/80-246 (attached).

Proposed Change:

The above options are in order of preference. If the feature is deemed so desirable that it cannot be removed, it must be made adequately robust.

Justification:

(1) Some compilers may have a serious problem distinguishing A from (A) in an actual parameter specification, nominally because of the use of bottom-up parsing mechanisms in the expression parser. While one may argue that marginal compiling techniques should not be encouraged, others may argue with as much right that runtime storage management mechanisms should be sufficiently robust to tolerate runtime specification of the activation record sizes.

(2) Consider the procedure WORKON defined to produce an array Y by some activity on the elements of array X, for example, transposition. With fixed array types, the procedure would look like:

```
type vector = array [1..50] of real;
...
procedure WORKON ( X: vector; var Y:vector);
    var i: 1..50;
begin
    for i := 1 to 50 do
       Y[i] := X[51 - i];
end;
. . .
```

and if A is a vector, then WORKON (A, A); can be expected to transpose A over itself correctly. But if a conformant array schema is used:

```
procedure WORKON (var X, Y : array [lo..hi: integer] of real);
```

then WORKON (A, A) will fail strangely, and WORKON ((A), A) is required.

The annoying thing about this failure is that the latter procedure is the one which is expected to be put in a source library to be copied into the user program and used without other than black-box

documentation. The problem with the proposed syntax is that the procedure cannot protect itself from misuse - it must depend on the caller to use it correctly. And yet the avowed intention of the construct in the first place was to permit the construction of procedure libraries which were essentially independent of the types in the calling program.

(3) The proposal eliminates a desirable implementation method. The proposal requires the calling program to allocate the copy of the variable, where for code economy the implementor may prefer the called program do so.

(4) A number of objections to conformant array parameters as previously specified still stand as objections to the current proposal:

(a) They emphasize structural compatibility of types, a phenomenon which is avoided in the theoretical studies of Pascal and in the draft proposed standard, in each case with great deliberation. Several other proposed modifications to permit structural compatibility of anonymous types have been firmly rejected on the basis of the importance of type identification and name-compatibility of types. This feature is deemed to be of such value that its consistency with such cherished characteristics of the language is of no consequence.

(b) Conformant array schemas provide no method of construction of a type-denoter for the types they represent. As a consequence, no related compatibilities can be specified, as between arrays and vectors, for example, and such compatibilities as may be required in a given procedure must be checked by user code at runtime.

(c) The expectation that many procedures using conformant array parameters will be included from source libraries creates a real problem in the intelligent use of the "ordinal-type-identifier" in the index-type-specification. Since such type-identifiers would in most cases be limits on the capabilities of the procedure, and would have to be source-included in a program which has no other use for them, it is likely that in the average installation the ordinal-type-identifier would usually degenerate to integer. Thus in most cases, any limitations the procedure really has must be protected by user code runtime checks.

(d) Because of the conformability rules, the use of "ordinal-type-identifier" doesn't prevent the system from having to perform runtime checks for the compatibility of index-type-specifications. Consider:

```
type rg10 = 1..10;  rg20 = 1..20;
var  A: array[rg10] of real;
     B: array[rg20] of real;
procedure P(VAR X: array [lo..hi:rg20] of real);

procedure Q(VAR Y: array [lo..hi:rg10] of real);
```

If procedure P contains the statement Q(X);
then P(A); is valid, but P(B) is invalid. And if the call on Q is conditional, e.g. if hi<=10 then Q(X);
then even P(B) is valid, but the proof is in the taste - you find out at run time. So the system has to perform the runtime check, or say that it doesn't, of course.

Comment on 6.6.3.7 Conformant-array-parameters (p. 37)

Status: Error

Problem:

The beginning of the paragraph following the first note on page 37 contains an elaborate specification which reduces to nothing of value. It contains at least one incorrect occurrence of "not" in "not a factor that is not a variable-access." It clearly does not represent the author's intent.

Proposed Change:

In the paragraph following the first note on page 37, delete the first sentence and the beginning of the second sentence up to "expression", and replace them with:
"The actual-parameter shall be an expression. If the actual-parameter is not a variable-access,"...

Justification:

The only English-language parse of the first sentence yields:
"The actual-parameter shall be either (a) a variable-access, or (b) an expression which is not denoted by a factor." (The clause "not a factor that is not a variable-access" translates to: "if it is a factor then it must be a variable-access", which is allowed by the first spec.)

Unfortunately, the only expressions allowed under (b) are those which contain relational-operators, adding-operators, or multiplying-operators, none of which can yield a value of array-type except by extension to the proposed standard. Moreover, the recommendation in the following note, that a "value" parameter can be constructed by the form "(A)" conflicts with the stated requirement, because the form "(A)" is a factor which is not a variable-access. So it is very unlikely that this restriction was intended as written.

It is not difficult to allow the generalization to "expression", since the conformability requirement will eliminate most possible productions and leave exactly three possibilities within the proposed standard: variable-access, character-string, and "(variable-access)". (It also allows any number of redundant parentheses around any of the three possibilities.) It is not clear whether the author intended to prevent character-string as a possibility, but it seems unnecessary to do so. Character-string parameters present no difficulty to the compiler-writer and considerable advantage to the user, whereas the form (A), which was clearly intended, causes additional headaches for the compiler-writer and the author of this standard.

It should also be noted that the generalization to "expression" allows implementations which support array arithmetic or array-valued functions to be included automatically without further local modifications to the conformant-array-parameter rules.

Comment on 6.6.3.7 Conformant-array-parameters (p. 37)

Status: Error

Problem: On page 37 in the second paragraph after the second note, beginning "If the actual-parameter is an expression whose value is denoted by a variable-access," the condition given is incorrect in two ways:

(1) The expression which is a variable-access is also an expression whose value is denoted by a variable-access. In this case the limitations should not be applied, since the rule above specifies that the parameter shall be passed "by reference" in this case.

(2) When the actual-parameter is an indexed-variable, the variable-access that is the actual-parameter is never the variable-access that closest-contains the conformant-array-parameter identifier -- the array-variable is.

Proposed Change:

At the end of the first paragraph of 6.6.3.7 (p.35), add:
"A parameter-identifier so defined shall be designated a conformant-array-parameter."

At the end of the paragraph at the top of page 37, just before the note, insert:
"The type denoted by the type-identifier contained by the conformant-array-schema in a conformant-array-parameter-specification shall be designated the fixed-component-type of the conformant-array-parameters defined by that conformant-array-parameter-specification."

Replace the second paragraph after the second note on page 37 with: "If the actual-parameter is not denoted by a variable-access and the actual-parameter contains an occurrence of a conformant-array-parameter, then for each occurrence of the conformant-array-parameter contained by the actual-parameter expression, either
(a) the occurrence of the conformant-array-parameter shall be contained by a function-designator contained by the actual-parameter expression, or
(b) the occurrence of the conformant-array-parameter shall be contained by an indexed-variable contained by the actual-parameter expression, such that the type of that indexed-variable is the fixed-component-type of the conformant-array-parameter.

Justification:

(1) If the actual-parameter is an expression whose value is denoted by a variable-access, it may be the form V, whereas the expression the author wants to limit has the form (V), because the former is passed by reference (and therefore is no problem), but the latter is passed by value, and its size must be known at compile-time.

(2) The idea is that if the actual-parameter contains a formal parameter from a higher-level activation and that formal parameter is itself a conformant-array-parameter, we want to be sure that we are not required to pass on something of unknown length, unless we can pass it by reference. Unfortunately, the variable-access which closest-contains the conformant-array-parameter is the variable-access of the array-variable of the indexed-variable of the component-variable of the variable-access which is the actual-parameter.

(3) Regretably, there is no good way to specify the particular syntactic entity which may not contain a conformant-array-parameter unless it is adequately subscripted. Consider the descent for (A[I]): expression, simple-expression, term, factor, (parens) expression, simple-expression, term, factor, variable-access, component-variable, indexed-variable, (a) array-variable, variable-access, entire-variable, variable-identifier, identifier; (b) (brackets) index-expression, expression,... It is easy to leap to the conclusion that indexed-variable is the target entity, but note the ancestral tree you have to give to distinguish the one you mean from the possible occurrence of another one in the index-expression.

The proposed change discards this approach in favor of a much more global, but apparently adequate, limitation. The weakness is that the proposed change assumes that there can be no legal operators on conformant-array-parameters per se, only on the fixed-component-type. Of course, it is always possible for the conformant-array-parameter to be passed to a function used in the computation of some value in the actual-parameter expression. So option (a) allows this, noting that the conformant-array-parameter will have to satisfy the usage constraints as an actual-parameter to that function.

(4) Note that the changes contain two insertions to define terms so that the restriction on actual parameters is comprehensible. They are not strictly necessary, but the existing wording for "conformant-array-parameter" requires an additional clause: "defining-occurrence for the block which contains the actual-parameter...". The existing (a) and (b) could be combined into a replacement for the proposed (b), and thus remove the need for defining "fixed-component-type", leaving as much of the existing text, and as little comprehensibility, as possible.

Comment on FOR statements

Status: Change.

Problem: DP7185/second edition changes the status from error to requirement in 6.8.3.9 for assigning-references within a for-statement. This may cause difficulties for some implementations. Consider

```
procedure p;
  var i: integer;  j: integer;
  function f: integer;
  begin
    f := 0;
    i := 1
  end;
begin
  for i := 1 to 10 do j := f
end
```

Without flow analysis or other relatively expensive mechanisms it is very difficult to detect the modification of i within f. This problem is very difficult in general and the space-overhead in compilation can be a burden.

Proposed Change: In 6.8.3.9, paragraph 2, replace sentence 3 with:
Neither the statement of a for-statement nor any procedure-and-function-declaration-part of the block that closest-contains a for-statement shall contain a statement threatening the variable denoted by the control-variable of the for-statement.

And a new paragraph to 6.8.3.9:

A statement S shall be designated as threatening a variable V if one or more of the following is true.

(a) S is an assignment-statement and V is denoted by the variable-access of S;
(b) S contains an actual variable parameter which denotes V;
(c) S is a procedure-statement that specifies the activation of the required procedure read or the required procedure readln, and V is denoted by an actual parameter contained by S;
(d) S is a for-statement and the control-variable of S denotes V.

Justification:

The present restrictions are unnecessarily complex and costly to enforce; as a consequence implementations are likely to not enforce them. It is preferable from the user's point of view that such parts of the language be enforced to promote the detection of programming errors and to avoid the creation of non-conforming programs. The proposed change is simpler to understand, more likely to be enforced, and in addition to the above advantages for users, allows the removal of run-time checks from for-statement loops.

Comment on section 6.8.2.3 (Procedure-statements) and section 6.9 (Input and output)

Status: Error

Problem Statement: The non-terminal symbols read-parameter-list, readln-parameter-list, write-parameter-list and writeln-parameter-list are never used in other syntax productions.

Proposed Change to the Draft Proposal:

In section 6.8.2.3 add the following to the end of the first paragraph:

The procedure-identifier in a procedure-statement containing a read-parameter-list shall denote the required procedure read; the procedure-identifier in a procedure-statement containing a readln-parameter-list shall denote the required procedure readln; the procedure-identifier in a procedure-statement containing a write-parameter-list shall denote the required procedure write; the procedure-identifier in a procedure-statement containing a writeln-parameter-list shall denote the required procedure writeln.

In the same section modify the definition of procedure-statement to read:

```
procedure-statement =
        procedure-identifier
            ( [ actual-parameter-list]|
              read-parameter-list|
              readln-parameter-list|
              write-parameter-list|
              writeln-parameter-list ) .
```

Comment on non-existence of applied occurrences

Status: Error

Problem: In subclause 6.2.2, the word identifier is used with (at least) four different meanings. In 6.2.2.1, it conforms to the (syntactic) definition given in 6.1.3. In 6.2.2.5, it refers to homonyms: two different syntactic identifiers having identical orthography but different derivations and meanings. In 6.2.2.7, there is the syntactic meaning as well as the meaning of homograph: having identical orthography. Then in 6.2.2.9 it is untenable. To correct it, remove all usages of identifier (and label) that conflict with the definition given in 6.1.3.
has to do with the set of x-identifiers. Such confusion is
Proposed change:

Replace 6.2.2.5 by

When an identifier or label has a defining-point for region A and another identifier or label having the same spelling has a defining-point for some region B enclosed by A, then region B and all regions enclosed by B shall be excluded from the scope of the defining-point for region A.

Replace 6.2.2.7 by

The scope of a defining-point of an identifier or label shall include no defining-point of another identifier or label having the same spelling.

In 6.2.2.8, change "all occurrences of that identifier or label shall be designated applied occurrences" to "each occurrence of an identifier or label having the same spelling shall be designated an applied occurrence of the identifier or label of the defining-point".

In 6.2.2.9, change "a type-identifier may have an applied occurrence in the domain-type" to "an identifier may have an applied occurrence in the type-identifier of the domain-type".

Justification: Without this change there are no applied occurrences.

Comment on File Handling Procedures (6.6.5.2, 6.9.2, 6.9.3, 6.9.4, 6.9.5)

Status: Error

Problem Statement:

Section 6.6.5.2 defines read(f,v) to be equivalent to:

```
        begin v := f^; get(f) end
```

and write(f,e) to be equivalent to:

```
        begin f^ := e; put(f) end
```

The proposed draft standard even contains a note making it clear that read is equivalent to the specified compound statement and not to a procedure whose body is the compound statement.

Consider the following variable declarations:

```
        var
            fa : array  [1 .. 10] of file of integer;
            ftext : array [0 .. 256] of text;
            a : array [1 .. 10] of real;
            i : integer;
            c : char;
```

The proposed Pascal standard leads one to believe that read(fa[i],i) is equivalent to:

```
begin i := fa[i]^; get(fa[i]) end
```

and that write(fa[fa[2]^],i) is equivalent to:

```
begin fa[fa[2]^]^ := i; put(fa[fa[2]^]) end
```

By choosing the proper values for the variables it's possible that the above read statement will read an integer value from the file buffer of one file but do the get operation on a different file. Likewise, the above write statement can do an assignment to the file buffer of one file but do the put operation on a different file. The above behavior is even more spectacular when textfiles are used. The proposed Pascal standard does not seem to adequately define the effect of:

```
readln(ftext[ ord(ftext[i]^)+ord(eoln(ftext[ord(c)])) ], i, a[i], c)
```

The Pascal file handling procedures should not be defined so that the file variable being accessed can change during the procedure execution.

Proposed Change to the Draft Proposal:

JPC believes that this is an important correction to the Pascal standard. However, the complexity of the issue precludes a reliable solution in the time allotted. The exact wording of the correction should be considered by ISO/TC 97/SC 5/WG 4. An example of an attempted correction follows:

In section 6.6.5.2 change the definition of read to:

Let f be a file-variable and vl...vn be variable-accesses then the procedure-statement read(f,vl,...,vn) shall access the file variable and establish a reference to that file variable for the remaining execution of the statement. The remaining execution of the statement shall be equivalent to

```
begin read(ff,vl); ... ; read(ff,vn) end
```

where ff denotes the referenced file variable.

Let f be a file-variable and v be a variable-access; then the procedure-statement read(f,v) shall access the file variable and establish a reference to that file variable for the remaining execution of the statement. The remaining execution of the statement shall be equivalent to

```
begin v := ff^; get(ff) end
```

where ff denotes the referenced file variable.

In section 6.6.5.2 change the definition of write to:

Let f be a file-variable and el...en be expressions; then the procedure-statement write(f,el,...,en) shall access the file variable and establish a reference to that file variable for the remaining execution of the statement. The remaining execution of the statement shall be equivalent to

```
begin write(ff,el); ... ; write(ff,en) end
```

where ff denotes the referenced file variable.

Let f be a file-variable and e be an expression; then the procedure-statement write(f,e) shall access the file variable and establish a reference to that file variable for the remaining execution of the statement. The remaining execution of the write statement shall be equivalent to

```
begin ff^ := e; put(ff) end
```

where ff denotes the referenced file variable.

In section 6.9.2 change subparagraph (a) to:

(a) read(f,vl,...vn) shall access the textfile variable and establish a reference to that textfile variable for the remaining execution of the statement. The remaining execution of the statement shall be equivalent to

```
begin read(ff,vl); ... ; read(ff,vn) end
```

where ff denotes the referenced textfile variable.

In section 6.9.2 change subparagraph (b) to:

(b) If v is a variable-access possessing the char-type (or subrange thereof), read(f,v) shall access the textfile variable and establish a reference to that textfile variable for the remaining execution of the statement. The remaining execution of the statement shall be equivalent to

```
begin v := ff^; get(ff) end
```

where ff denotes the referenced textfile variable.

In section 6.9.2 change the first sentence of subparagraph (c) to:

(c) If v is a variable-access possessing the integer-type (or subrange thereof), read(f,v) shall access the textfile variable and establish a reference to that textfile variable for the remaining execution of the statement. The remaining execution of the statement shall cause the reading from the referenced textfile variable of a sequence of characters.

In the last sentence of subparagraph (c) of section 6.9.2 change "the buffer-variable f^ does not" to "the buffer-variable of the referenced textfile does not"

In section 6.9.2 change the first and last sentences of subparagraph (d) similiarly to the change of subparagraph (c).

In section 6.9.3 change the definition of readln to:

Readln(f,vl,...,vn) shall access the textfile variable and establish a reference to that textfile variable for the remaining execution of the statement. The remaining execution of the statement shall be equivalent to

```
begin read(ff,vl,...,vn); readln(ff) end
```

where ff denotes the referenced textfile variable.

In section 6.9.4.1 change the definition of write to:

Write(f,pl,...,pn) shall access the textfile variable and establish a reference to that textfile variable for the remaining execution of the statement. The remaining execution of the statement shall be equivalent to

```
begin write(ff,pl); ... ; write (ff,pn) end
```

where ff denotes the referenced textfile variable.

In section 6.9.5 change the definition of writeln to:

Writeln(f,p1,...,pn) shall access the textfile variable and establish a reference to that textfile variable for the remaining execution of the statement. The remaining execution of the statement shall be equivalent to

begin write(ff,p1,...,pn); writeln(ff) end

where ff denotes the reference textfile variable.

Schema Array Proposal

USA Contribution on Schema Arrays for Pascal

Abstract

This proposal introduces a new concept into Pascal - the schema. Once defined it solves the same problem that conformant arrays attempted to address. The principle advantage with this mechanism is that it provides a broader base on which to build; it resolves many of the problems found with conformant arrays and offers the opportunity to provide other features in the future should the need be determined.

The problem addressed by conformant arrays is one of how to pass arrays into a procedure or function in such a way that the bounds of the array are provided by the actual parameter - rather than by the formal parameter. This function is very desirable in the context of being able to write generic procedures and functions.

This proposal will be based upon X3J9/80-192 with references to conformant arrays omitted.

Overview

A schema can be thought of as a collection of types; each member of the collection is related to the other members in that they each have the same overall structure. The structure of each type is that of an array with the same component type. However, each array has a different index-type.

We permit a parameter of a procedure or function to specify that it will accept any actual parameter whose type is a member of a specified schema. In this way we permit the procedure or function to operate on a number on values with different types, although only from the same schema.

Proposal

In section 6.2.1 modify the production for type-definition-part:

```
type-definition-part =
       [ "type" ( type-definition | schema-definition ) ";"
       { ( type-definition | schema-definition ) ";" } ] .
```

Effect

This says that the type-definition-part of a block is composed of any number of type and schema definitions.

Modify the production in section 6.4.1 for a new-type:

```
new-type = new-ordinal-type | new-structured-type |
       new-pointer-type | discriminated-schema .
```

Effect

This specifies that a new-type may be created by any of the existing means in Pascal or by selecting one of the members of a schema.

Add a section between 6.4 and section 6.5:

6.x Schema-definitions

6.x.1 General. A schema-definition shall introduce an identifier to denote a schema. A schema defines a collection of new-types whose type-denoter is a discriminated-schema.

```
schema-definition =
       identifier formal-discriminant-part  "="  array-schema .

formal-discriminant-part =
       "(" discriminant-specification
       { ";" discriminant-specification } ")" .

discriminant-specification =
       identifier-list  ":"  ordinal-type-identifier .

array-schema = [ "packed" ] "array" "[" schema-index-type
       { ";" schema-index-type } "]" "of" component-type .
```

```
schema-index-type = ( constant | discriminant-identifier )
      ".." ( constant | discriminant-identifier ) .

discriminant-identifier = identifier.

schema-identifier = identifier.
```

The occurrence of an identifier in a schema-definition of a type-definition-part shall constitute its defining-point for the region that is a block. Each applied occurrence of that identifier shall denote the same schema. Except for applied occurrences of the identifier in a discriminated-schema as the domain-type of a pointer-type, the schema shall not contain an applied occurrence of the schema-definition.

Effect

The above definitions add the mechanism by which to define a schema. The leading identifier on the schema-definition (schema-identifier) becomes known. A schema may not have any references to itself except when used as the domain of a pointer; and in that case, it must only be used with the actual-discriminants (discriminated-schema). Thus, a schema has the same scope as a type declared at the same place.

Add a section after 6.x.1

6.x.2 Formal-discriminant-part. The formal-discriminant-part in a schema-definition shall define the formal-discriminants. The occurrence of a identifier in a discriminant-specification shall constitute its defining point as a discriminant-identifier for that region of the program that is the following array-schema.

For every discriminant-identifier in formal-discriminant-part, there shall be at least one applied occurrence in the array-schema. The occurrence of a discriminant-identifier in a schema-index of an array-schema shall specify that there is one type-denoter which is a member of the schema for each allowed value of the discriminant-identifier such that all other schema-index values in the schema are the same.

Note: this implies that the number of type-denoters in the domain of the schema is the product of the number of values for each occurrence of each discriminant-identifier.

Effect

The formal-discriminant-part is used to associate identifiers with the schema so that the domain (members of the schema) can be determined. Every identifier used in the formal-dsicriminant must be used at least once in the following array-schema. In the following example, SmallVect is a collection of ten type-denoters with index-types "0..1", "0..2", ... , "0..10".

```
type
  SmallInt = 1 .. 10;
  SmallVect(HighBound : SmallInt) =
        array [ 0 .. HighBound ] of Real;
```

Add a section after 6.x.2

6.x.3 Discriminated-schema. A discriminated-schema selects one of the members of a schema as a new-type. The discriminant-values are bound to their corresponding discriminant-specifications in the formal-discriminant-part for the schema. The number of discriminant values must be equal to the number of formal-discriminants and each value must be assignment compatible with the type of the corresponding formal-discriminant.

```
discriminated-schema = schema-identifier actual-discriminant-part .

actual-discriminant-part = "(" discriminant-value
      { "," discriminant-value } ) .

discriminant-value = constant .
```

Any schema designated packed and denotes an array-schema having its schema-index-type specifying its smallest value a constant whose value is 1, and having as its component-type a denotation of the char-type, shall be a string-schema. Any new type specifying a discriminated-schema which is a string-schema shall be designated a string-type.

Effect

A discriminated-schema is a type-denoter selected from the collection of type-denoters in the schema. The values given in the actual-discriminant-part are used (substituted) for the formal-discriminants in the array-schema. Thus the discriminated-schema: "SmallVect(7)" selects the member of the schema which is equivalent to (but not the same as) the array:

```
array [ 0 .. 7 ] of Real
```

An attempt to specify the schema as "SmallVect(11)" will result in an error because the value 11 is not assignment-compatible with the type of HighBound.

It must be noted that although a discriminated-schema is equivalent in structure to an array-type, it never the same (in the sense of type compatibility). Moreover, two discriminated-schemas that specify the same discriminant-values are not the same. In the following fragment V2 and V3 have the same type, and V4, V6 and V7 have the same type.

```
type
  T1      = SmallVect(3);
  T2      = SmallVect(3);
  T3      = T1;
var
  V1      : SmallVect(3);
  V2,V3   : SmallVect(3);
  V4      : T1;
  V5      : T2;
  V6      : T1;
  V7      : T3;
```

Modify the production in section 6.6.3.1

```
formal-parameter-section =
      value-parameter-specification  |
      variable-parameter-specification  |
      constant-parameter-specification  |
      procedural-parameter-specification  |
      functional-parameter-specification .
```

Effect

This introduces constant-parameter-specification.

Modify the production in section 6.6.3.1

```
variable-parameter-specification =
      "var"  identifier-list  ":"
      (type-identifier | schema-identifier) .
```

Effect

The modified production states that a variable may be passed into a procedure or function whose type-denoter is a member of a schema. When a schema-identifier is specified, then the parameter may be of any type which is a member of the schema.

Add this production to section 6.6.3.1

```
constant-parameter-specification  =
      "const"  identifier-list  ":"  schema-identifier .
```

Effect

A constant-parameter-specification is permitted only to be used with schemas and permits literal character-strings to be passed efficiently to a procedure or function. It also permits variables which are array-schemas to be passed as "read-only" variables. It should be possible to extend this concept to other types in the future if it found to be desirable.

Add this to the text of section 6.6.3.1

The occurrence of an identifier in in the identifier-list of a constant-parameter shall constitute its defining point as a read-only-variable for the region that is the block, if any, of which it is a formal-parameter.

Effect

All parameters that are specified with the constant mechanism are identified as being read-only varaibles, this permits them to be limited to being factors within the block.

Add to section 6.6.3.3

If the formal parameters are specified in a variable-parameter-specification in which there is a schema-identifier, the type possessed by the actual-parameter shall be a discriminated-schema designating the same schema-identifier as the formal parameter or the actual-parameter shall be itself a parameter that was specified with the same schema-identifier; and the type possessed by the formal-parameter shall be distinct from any other type.

Effect

This states that a formal parameter that was declared with a schema will only permit the actual parameter to be of type which is part of the same schema. A formal-parameter which is a schema may in turn be passed to as a variable-parameter utilizing the same schema.

If the form of the parameter list includes an identifier-list, then all the actual parameters must be of the same type: this is true for schemas as well as other types.

The following example adds two vectors, element by element, and returns the result in the first parameter.

```
procedure AddVectors(var A,B,C : SmallVect);
var
  i  : natural;
begin
  for i := 0 to B.HighBound do
    A[i] := B[i] + C[i]
end;
```

Add a section between 6.6.3.3 and 6.6.3.4

6.6.3.y Constant parameters. The actual-parameter shall be an expression. The formal parameters that occur in a single constant-parameter-specification shall possess an array-type which is distinct from any other type. The type possessed by the actual-parameter shall be a discriminated-schema designating the same schema-identifier as the formal parameter or the actual-parameter shall be itself a parameter that was specified with the same schema-identifier; or the actual-parameter must be a string-type and the formal parameter must designate a string-schema.

For an actual-parameter that denotes a variable-access, there shall be no assigning-reference during the activation of the block of procedure or function to the actual-parameter.

Effect

This introduces a parameter mechanism into Pascal that permits may not be altered during the activation of the associated procedure or function. Any expression may be specified by the actual parameter, however the only expression that is not a variable-access will be a string literal. Thus, the mechanism achieves not only protection of the actual-parameter but also permits literal strings to be specified.

The method of passing the parameter may be chosen by the implementation, one suitable method may by passing an indirect reference in the parameter list.

Modify the production in 6.7 for a factor

```
factor = variable-access       | unsigned-constant  |
         function-designator   | set-constructor    |
         "(" expression ")"    | "not" factor       |
         schema-discriminant   | read-only-variable .

schema-discriminant = parameter-identifier
         "." discriminant-identifier .

read-only-variable = variable-access .
```

Effect

Addition to factor is used to indicate that a factor may also be a schema-discriminant.

Add the production in 6.7 for a schema-discriminant

```
schema-discriminant = variable-access
         "." discriminant-identifier
```

Effect

A schema-discriminant is used to determine that actual-discriminants of the the actual-parameter. Because a factor can never appear as a target of an assignment, the discriminant may never be altered. The value of the discriminant could be thought of as a "read-only" value associated with the variable (or parameter).

Example

```
const
  MaxMatrix  = 100;

type
  Positive    = 1..MaxMatrix;
  Matrix(M,N : Positive) =
      array[ 1..M, 1..N ] of Real;
  Square(Len : Positive) = Matrix(L,L);

procedure Transpose ( var M : Square );
  var
    I,J  : Positive;
    R    : Real;
  begin
    for I := M.Len downto 2 do
      for J := I-1 downto 1 do
        begin
          R := M[I,J]
          M[I,J] := M[J,I]
          M[J,I] := R
        end
  end;
```

IMPLEMENTATION NOTES ONE PURPOSE COUPON

0. **DATE**

1. **IMPLEMENTOR/MAINTAINER/DISTRIBUTOR** *(* Give a person, address and phone number. *)*

2. **MACHINE/SYSTEM CONFIGURATION** *(* Any known limits on the configuration or support software required, e.g. operating system. *)*

3. **DISTRIBUTION** *(* Who to ask, how it comes, in what options, and at what price. *)*

4. **DOCUMENTATION** *(* What is available and where. *)*

5. **MAINTENANCE** *(* Is it unmaintained, fully maintained, etc? *)*

6. **STANDARD** *(* How does it measure up to standard Pascal? Is it a subset? Extended? How.*)*

7. **MEASUREMENTS** *(* Of its speed or space. *)*

8. **RELIABILITY** *(* Any information about field use or sites installed. *)*

9. **DEVELOPMENT METHOD** *(* How was it developed and what was it written in? *)*

10. **LIBRARY SUPPORT** *(* Any other support for compiler in the form of linkages to other languages, source libraries, etc. *)*

(FOLD HERE)

PLACE
POSTAGE
HERE

Bob Dietrich
M.S. 92-134
Tektronix, Inc.
P.O. Box 500
Beaverton, Oregon 97077
U.S.A.

(FOLD HERE)

NOTE: Pascal News publishes all the checklists it gets. Implementors should send us their checklists for their products so the thousands of committed Pascalers can judge them for their merit. Otherwise we must rely on rumors.

Please feel free to use additional sheets of paper.

IMPLEMENTATION NOTES ONE PURPOSE COUPON

POLICY: PASCAL USERS GROUP (15-Sep-80)

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 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 goals and is being used successfully for:

* teaching programming concepts
* developing reliable "production" software
* implementing software efficiently on today's machines
* writing portable software

Pascal implementations exist for more than 105 different computer systems, and this 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.
> Springer-Verlag Publishers: New York, Heidelberg, Berlin
> 1978 (corrected printing), 167 pages, paperback, $7.90.

Introductory textbooks about Pascal are described in the "Here and There" 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.

Remember, **Pascal User's Group** is each individual member's group. We currently have more than 3500 active members in more than 41 countries. this year Pascal News is averaging more than 100 pages per issue.

Policy