

# 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 ?

EX LIBRIS: David T. Craig  
736 Edgewater  
[# \_\_\_\_\_ ] Wichita, Kansas 67230 (USA)

- \* 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.

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

**\*\*Note\*\***

- We will not accept purchase 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.   |
|--------------------------|-------------------------------|-------------------|--------|---------|
| <input type="checkbox"/> | Enter me as a new member for: | [ ] 1 year \$10.  | \$14.  | A\$ 8.  |
| <input type="checkbox"/> | Renew my subscription for:    | [ ] 2 years \$18. | \$25.  | A\$ 15. |
|                          |                               | [ ] 3 years \$25. | #35.   | A\$ 20. |
| <input type="checkbox"/> | 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 1' 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.

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

|   |
|---|
| Mail request to:<br><br>ANPA/RI ..<br>P.O. Box 598<br>Easton, Pa. 18042<br>USA<br>Attn: R.J. Cichelli |
|---|

Office use only

Signed \_\_\_\_\_  
Date \_\_\_\_\_

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

# Index

PASCAL NEWS #21

APRIL, 1981

INDEX

|    |  |
|----|--|
| 0  | POLICY, COUPONS, INDEX, ETC.   |
| 1  | EDITOR'S CONTRIBUTION  |
| 3  | HERE AND THERE WITH Pascal   |
| 3  | Book review: "The Pascal Handbook"   |
| 4  | Book review: "Introduction to Pascal"  |
| 5  | Tidbits  |
| 5  | PUG PRESS ... our sister publication?  |
| 6  | I'm not sure??   |
| 7  | APPLICATIONS   |
| 7  | The EM1 compiler -- Andrew s. Tanenbaum.                                       |
| 23 | Unreal Arithmetic -- Jeff Pepper.  |
| 27 | ARTICLES   |
| 27 | "An extention to Pascal Read and Write Procedures"<br>-- by David Rowland.     |
| 28 | "PDP-11 Pascal: The Swedish Compiler vs. OMSI Pascal-1"<br>-- by Margret Kulos |
| 40 | OPEN FORUM FOR MEMBERS   |
| 43 | PASCAL STANDARDS   |
| 85 | ONE PURPOSE COUPON, POLICY   |

---

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 our 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 on 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 succinct 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 J6W/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 UCSD 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.

Secondly, 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.

## \* Congratulations - On Your Newly Acquired 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 monetary 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, Wis. 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 categories will be: most unusual, most beautiful, most interesting, most common, and most humorous. 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

Page 2

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 Sherwood Pet in St. Paul. He would like a male pet PUG at a reasonable price.

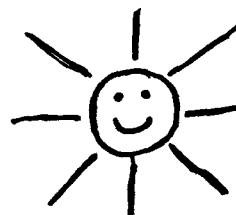
Eunice Thorson, 536 1st St., Proctor, Mn. 55810; recently lost her fourteen year old PUG. She would like another girl puppy or older PUG.

## \* 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



PASCAL NEWS #21

APRIL, 1981

PAGE 5

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, under-water exploits; shaming-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, bicycle loops, orchards, cottages, mist gardens, arboretum/terminal stands, geodesic 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 correspondence to me at: Kevin Switzer, 1534 Ford, Lincoln Park, Michigan 48146.

\* \*\*\*\* \* \* \* \* \* \* \* \* \* \* \* \*

Pascal User's Group  
% Rick Shaw  
Box 888524  
Atlanta, Georgia 30338

# Applications

## EP-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.

MOL <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

AD1 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

CM1 - 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, LGS  
 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 + LGS  
 LDL x = LAL x + LGI 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:  
 DMU = DUP 8 + DDV + DMU + DSB

GROUP 6:  
 ADI m = LOC m + PAD

GROUP 7:  
 INC = LOC 1 + ADD  
 INL x = LQL x + INC + STL x  
 INE x = LOE x + INC + STE x  
 DEC = LOC 1 + SUB  
 DEL x = LQL 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:  
 BRN 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 = LQL 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 = LQL 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 }
```

```
stard = 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; EMOM = 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;

```

```

instr: mnem;      { holds the instruction number }
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:=memm(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:=memm(sp); newsp(sp-2) end;

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

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

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

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

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

```

```

if size=1
  then push(memw(a))
  else if odd(size) or (size<=0)
    then trap(E00DZ)
    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,memw(sp)); newsp(sp-2) end
    else if odd(size) or (size<=0)
      then trap(E00DZ)
      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:bfstype; 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;
      end;
    end;
end;

```

```

xorff: if bit(i,w1)+bit(i,w2) = 1 then j:=j+1
end
end;
if j <= t15a1 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(E00DZ);
  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 dodsbs(ds,dt:double):double;
begin { subtract two doubles } dodsbs:=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 dofdiv(x,y:real):real;
begin { divide two reals } dofdiv:=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+pdsiz*ueerrorproct+pdbase));
  if n in [ESTACK,EHEAP,EILLINS,EODDZ,ECASE,ECAL,ENEMFLT,EPTR,
           EPROC,EPC]
    then goto 9999;
end;

procedure dortt;
var s:adr;
begin
  newpc(memi(lb-reta)); s:=lb-mrksize-2; newlb(memi(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,l,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;
        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
end;
{ read descriptor table }
pd:=header[TEXT];
for i:=1 to header[NPROC]*pdsiz 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 + pdsiz*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 LQL, 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 LQL with a one byte offset, and LQL 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 LQL 0, LQL 2, LQL 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 (LQL 0, LQL 2, LQL 4, etc.) in some register, and have all the LQL mini format instructions branch to a common routine. This routine can then determine the operand by subtracting the code for LQL 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 )

```

-----)
(                               )
(                               )
(                               )
-----)

```

```

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
end
end
else
begin insr:=CAL; k:=opcode-cutoff end;

```

```

-----)
(                               )
(                               )
(                               )
-----)

```

```

case insr of
NON: trap(EILLINS);
{ LOAD GROUP }
LOC: push(k);
LNC: push(-k);
LOL: push(memu(lb+argx(k)));
LOE: push(memu(argx(k)));
LOP: push(memu(memu(lb+argx(k))));
LAI: begin k:=argx(k); a:=popa; b:=memu(a); store(a,b+k); pushx(k,b) end;
LOF: push(memu(popa+k));
LAL: push(lb+argx(k));
LAE: push(argx(k));
LEX: begin a:=lb; for j:=1 to argn(k) do a:= memu(a-stad); push(a) end;
LOI: pushx(argx(k),popa);
LOS: begin k:=popa; pushx(argx(k),popa) end;
LDL: begin k:=argx(k); push(memu(lb+k)); push(memu(lb+k+2)) end;
LDE: begin k:=argx(k); push(memu(k)); push(memu(k+2)) end;
LDF: begin a:=popa; push(memu(a+k)); push(memu(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:=argx(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(argx(k),popa);
STS: begin k:=popa; popx(argx(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(dofdvd(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) ;
        push( 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 }
INW,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*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 programmer. These mnemonics are MRK and MRS, corresponding to the immediate and stacked forms respectively. The decision about using alternate context 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-stdt);
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,memu(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+pdsiz*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+pdsiz*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));
LMI: 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 W. 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;<br>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 UUnequal (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;
  UUnequal := 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.byf[loc] > b.byf[loc] THEN state := bigger
        ELSE IF a.byf[loc] < b.byf[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;
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' 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] <= 256
          THEN num.byf[i] := tmp[i]
          ELSE
            BEGIN
              tmp[i+1] := tmp[i+1] + tmp[i] DIV 256;
              num.byf[i] := tmp[i] MOD 256
            END;
        END;
    END;
END;

```

```

{check for high byte overflow}
IF tmp[byteMax] <= 256
  THEN num.byf[byteMax] := tmp[byteMax];
  ELSE ErrorTrap ('input too large ');
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.byf[0], digits);
  FOR i := 0 to 2 DO s[i] := digits[i];

  {1st byte -- multiply by 256, add to s}
  GetDigits (num.byf[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.byf[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.byf[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.byf[4] > 0 THEN
    BEGIN
      GetDigits (num.byf[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;
        END;
      END;
    END;
END;

```

```

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.by[5] > 0 THEN
  BEGIN
  GetDigits (num.by[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;
END;

*** IF YOU INCREASE THE NUMBER OF BYTES BEYOND 0..6: repeat the process
as above for a) 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;
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;
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
  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.by[i] + b.by[i];
  FOR i := 0 TO byteMax - 1 DO
    IF tmp[i] <= 255
      THEN c.by[i] := tmp[i]
      ELSE
      BEGIN
      c.by[i] := tmp[i] - 256;
      tmp[i+1] := tmp[i+1] + 1
      END;
    IF tmp[byteMax] <= 255
      THEN c.by[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
  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.by[i];
  FOR i := 0 TO byteMax - 1 DO
    IF tmp[i] >= b.by[i]
      THEN c.by[i] := tmp[i] - b.by[i]
      ELSE
      BEGIN
      c.by[i] := tmp[i] + 256 - b.by[i];
      tmp[i+1] := tmp[i+1] - 1
      END;
    c.by[byteMax] := tmp[byteMax] - b.by[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.by[i-j] * b.by[j]);
    END;
  FOR i := 0 TO byteMax - 1 DO
    IF tmp[i] <= 255
      THEN c.by[i] := tmp[i]
      ELSE
      BEGIN
      c.by[i] := tmp[i] MOD 256;
      tmp[i+1] := tmp[i+1] + (tmp[i] DIV 256)
      END;
    IF tmp[byteMax] <= 255
      THEN c.by[byteMax] := tmp[byteMax]
      ELSE ErrorTrap ('mult overflow ');
    c.pos := (a.pos AND b.pos) OR NOT (a.pos OR b.pos);
  END;
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.by[asize] = 0) AND (asize > 0) DO asize := asize - 1;
    bsize := byteMax;
    WHILE (b.by[bsize] = 0) AND (bsize > 0) DO bsize := bsize - 1;
    IF asize = bsize
      THEN TooFar := TRUE
      ELSE
        BEGIN
          FOR i := byteMax downto 1 do shifted.by[i] := b.by[i-1];
          shifted.by[0] := 0;
          TooFar := UUGreater (shifted, a);
        END;
      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.by[i] := 0; {initialize all 0's}

      shiftCt := 0;
      WHILE NOT TooFar (a,b) DO
        BEGIN
          FOR i := byteMax DOWNT0 1 DO b.by[i] := b.by[i-1]; {shift left}
          b.by[0] := 0;
          shiftCt := shiftCt + 1;
        END ;
        FOR i := shiftCt DOWNT0 0 DO
          BEGIN
            WHILE NOT UUGreater (b,a) DO
              BEGIN
                q.by[i] := q.by[i] + 1;
                USub (a,b,a);
              END;
            IF i > 0 THEN
              BEGIN
                FOR j := 0 to byteMax - 1 DO b.by[j] := b.by[j+1]; {shift right}
                b.by[byteMax] := 0;
              END;
            END;
            rem.by := a.by;
          END;
        END;
      END;
    END;
  {-----}

  procedure Main;
  VAR a,i,f: integer;
  x,y,z,rem: unreal;
  c1: 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');
  
```

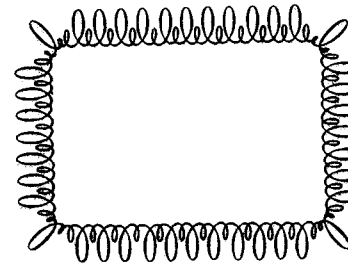
```

    '=': IF UEqual(x,y) THEN write ('equal') ELSE write ('not equal');
    'c': BEGIN dummy := Uconvert(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:16);
  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.

## PASCAL INPUT/OUTPUT

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 implementors in providing a correct implementation of "standard" Pascal in an effort to improve the portability 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-> 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 operaoonr 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  $1.0 + \text{EPS} < 1.0$ ) 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 delimiters 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.

