PASCAL USERS GROUP

Pascal News

NUMBER 17

COMMUNICATIONS ABOUT THE PROGRAMMING LANGUAGE PASCAL BY PASCALERS

MARCH, 1980

POLICY: PASCAL NEWS (17-Mar-80)

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

* Pascal News contains all we (the editors) know about Pascal; we use it as the vehicle to answer all inquiries because our physical energy and resources for answering individual requests are finite. As PUG grows, we unfortunately succumb to the reality of:

  1. Having to insist that people who need to know "about Pascal" join PUG and read Pascal News - that is why we spend time to produce it!

  2. Refusing to return phone calls or answer letters full of questions - we will pass the questions on to the readership of Pascal News. Please understand what the collective effect of individual inquiries has at the "concentrators" (our phones and mailboxes). We are trying honestly to say: "We cannot promise more that we can do."

* Pascal News is produced 3 or 4 times during an academic year; usually in September, November, February, and May.

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

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

* Pascal News is divided into flexible sections:

  POLICY - explains the way we do things (ALL-PURPOSE COUPON, etc.)

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

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

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

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

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

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

- - - - - - ALL-PURPOSE COUPON - - - - - - (17-Mar-80)

Pascal User's Group, c/o Rick Shaw
Digital Equipment Corporation
5775 Peachtree Dunwoody Road
Atlanta, Georgia 30342 USA

NOTE

- Membership is for an academic year (ending June 30th).
- Membership fee and All Purpose Coupon is sent to your Regional Representative.
- SEE THE POLICY SECTION ON THE REVERSE SIDE FOR PRICES AND ALTERNATE ADDRESS if you are located in the European or Australasian Regions.
- Membership and Renewal are the same price.
- The U. S. Postal Service does not forward Pascal News.

- -

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

[ ] Renew my subscription for:

[ ] Send Back Issue(s) _____

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

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

[ ] Comments: _____

_____

_____

ENCLOSED PLEASE FIND: $ / A$ / £ ____.____

NAME _____

ADDRESS _____

_____

_____

PHONE _____

COMPUTER _____

DATE _____

JOINING PASCAL USER'S GROUP?

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

- American Region (North and South America): Send $6.00 per year to the address on the reverse side. International telephone: 1-404-252-2600.
- European Region (Europe, North Africa, Western and Central Asia): Join through PUG (UK). Send £4.00 per year to: Pascal Users Group, c/o Computer Studies Group, Mathematics Department, The University, Southampton SO9 5NH, United Kingdom; or pay by direct transfer into our Post Giro account (28 513 4000); International telephone: 44-703-559122 x700.
- Australasian Region (Australia, East Asia - incl. Japan): PUG(AUS). Send $A8.00 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 and European Regions must join through their regional representatives. People in other places can join through PUG(USA).

RENEWING?

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

ORDERING BACK ISSUES OR EXTRA ISSUES?

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

SENDING MATERIAL FOR PUBLICATION?

- Your experiences with Pascal (teaching and otherwise), ideas, letters, opinions, notices, news, articles, conference announcements, reports, implementation information, applications, etc. are welcome. Please send material single-spaced and in camera-ready (use a dark ribbon and lines 18.5 cm wide) form.
- All letters will be printed unless they contain a request to the contrary.

Contributors to this issue (#17) 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
Administration	Moe Ford, Kathy Ford, Jennie Sinclair

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.

X Distribution charge: $50.00

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

Mail request to:

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

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

( ) ANSI-Standard

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

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

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

Office use only

Signed _____
Date _____

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

Editor's Contribution

GETTING STARTED

Let me start my first editorial by saying, "I can't believe how hard this job is!!" My esteem for Andy Mickel has always been high, but after the last few months, it has gone up astronomically! I don't know how one person had all the time--there are so many things to do, and I have been lucky enough to have alot of help.

My section editors have been very prompt (for the most part!) and have made the job "do-able". And, I might add, PUG has hired some part-time clerical help that is out of this world! To round it off, the switch to a commercial printer (oh, the luxury of a university print shop) has been quite successful. I could not ask for better service. Their prices are close to those we paid in the past.

My thanks
must go to the membership, who have been so patient with me. This issue represents a tremendous learning curve for me (and culture shock!). Things will go smoother starting next issue.

NEXT ISSUE (#18) - SPECIAL!!

Speaking of next issue, we at PUG are pleased to announce that the next one will be completely devoted to the ISO Draft Standard for Pascal. (See Jim Miner's article this issue for a discussion of this and other items concerning standards.)

We are currently preparing this document for reproduction; it will be out no later than one month after this issue (#17).

ABOUT THIS ISSUE

WOW!! Is there alot of good stuff in this issue! Pascal has been on everyone's tongue lately, so "Here and There" is chock full of "newsy" information. We also have a large number of books and articles that have been reviewed this quarter, as well as an excellent in-depth review of the text Alagic and Arbib by one of our readers. (We could use more contributions such as this.)

The "Articles" section is kicked off by lucid discussion of "Conformant Array Parameters" authored by Arthur Sale (who else!). This article is highly recommended for review by all readers because of its controversial, proposed inclusion into the ISO standard.

There is no lack of contributions to the "Software Tools" section either. Nearly one-quarter of the issue is devoted to publishing programs and algorithms. This quarter many checklists are included in the "Implementation Notes" section, as well as some contributions to ur new section, "Validation Suite Reports".

A great deal of fine work went into this issue. We hope you like it.

Rick

Here and There With Pascal

```
TTTTTTT
   T
   T
   T
   T
   T
   T  IDBITS
```

J. Mack Adams, Comp. Sci Dept., Box 3CU, New Mexico State University, Las Cruces, NM 88001: "We have added an assertional checking capability to UCSD Pascal and have developed a debugging system based on assertional checking and symbolic execution. A paper on the system will be presented at ACM 79..." (*79/05/14*)

Ron Barstad, P.O. Box 6000, B-118, Phoenix, AZ 85005: "The Pascal on the (*USW Louisiana*) L68 (Multics) is only a subset. The L66 version from Waterloo is a full blown batch and/or TSS version. (*79/09/14*)

Dr. Oddur Benediktsson, Science Institute, University of Iceland, Dunhaga 3, Reykjavik: "We...are looking for a PASCAL compiler for...our PDP-11 RSX-11M system and so far have found only the OMSI product which we find a bit on the expensive side at $1500. We would also rather have the P-code type compiler if available. Can you make any suggestions? (*78/11/23*)

Rick Boggs, Nationwide Insurance, One Nationwide Plaza, Columbus, OH 43216: "Our problem is one of finding a Pascal implementation which matches our operating environment: a large-scale IBM/AMDAHL center running MVS 3.7 and...both the TSO and VSPC interactive systems." (*79/10/10*)

Paul C. Boyd, PPG Industries, Box R, Elwin-Mt. Zion Rd., Mt. Zion, IL 62549: "We are hoping to implement the OMSI PASCAL-1 package on a DEC PDP-11/34...under RSX-11/M...to develop process control programs to run on a network of DEC LSI-11/23 micros.... I would appreciate hearing from any OMSI PASCAL-1 users with experience in digital control applications." (*79/09/27*)

Glenn A. Burklund, 3903 Carolyn Ave., Fairfax, VA 22031: "Have North Star (UCSD) Pascal----it is miserable. Going Pascal/Z...for scientifc and engineering applications. The funct. & proc. are th main features of interest. It is virtually aimpossible to implement under North Star Pascal. Unless it is practical to implement these calls easily, Pascal will wither on the vine." (*79/10/09*)

John D. Bush, Minnesota Power & Light Co., 30 West Superior St., Duluth, MN 55802: "I have been trying to get programmers and DP Managers at MP&L interested in Pascal. By finding compilers for our Prime and IBM machines, I hope to give some of these people a chance to experiment with the language." (*79/10/03*)

Jim Carlson, School of Dentistry, University of the Pacific, 2155 Webster St., San Francisco CA 94115: "The School of Dentistry has recently acquired an Omsi Pascal Compiler...configured to operate under RSX-11M and will be installed on a PDP-11/34. We plan to use Pascal primarily for administrative purposes, but it will also be available for uses in other areas." (*79/05/22*)

M. B. Clausing, 5603 Fisher Dr., Dayton, OH 45424: "If the matter's still at issue, I vote not to affiliate with ACM. I see no particular advantage." (*79/07/06*)

John Corliss, Loyola University of Chicago, 6525 N. Sheridan Road, Chicago, IL 60026: "Loyola University...has acquired the Pascal compiler from the University of Manitoba for academic instructional use...we are (*interested*) in acquiring PASCAL subroutine libraries that we could use in our computer science classes." (*79/05/14*)

Don R. Couch, 5100 Montreal Dr., San Jose, CA 95130: "I am a student in a Cogswell College Pascal course, and use Pascal on a PDP-11/10 computer at American Microsystems, Inc." (*no date*)

R. H. Frank, Digital Consulting Corporation, P.O. Box 32505, San Jose, CA 95152: "Our company has just released a Pascal Compiler (P2 derivative) for the popular CP/M microcomputer system." (*79/09/26*)

Jim Gagne, M.D., Datamed Research, 1433 Roscomare Rd., Los Angeles, CA 90024: "Who's your medical applications editor (if any)? I'll do it if you need." (*79/05/30*)

Anton L. Gilbert, Information Sciences, U.S. Army White Sands Missile Range, NM 88002: "I am a new Pascal users. It will be used in my research group...on a PDP-11/70, PDP-11/35, a PDP-11/34 (* all under RSX-11M) and a PDP-11/15 (RT-11). One of my employees...is especially interested in Pascal in Image Processing Research." (*79/06/12*)

Ricardo O. Giovannone, Box 3606, University Park Branch, Las Cruces, NM 88003: "I am a graduate student at New Mexico State University...using this language since fall '78 and I really like it.... At the moment, I am working in a project dealing with implementation of an Educational Data Base System using Pascal as a host language. ...We hope to finish in this fall. We are using UCSD Pascal Version I.4." (*79/08/20*)

Mark Gordon, Computer Business Systems, Box 421, Truro, Nova Scotia B2N 5C5: "I am using a DEC PDP-11 under RSTS/E". (*79/05/23*)

Roedy Green, 1478 East 27th Avenue, Vancouver, British Columbia V5N 2W5: "I'm loking after a computer acquisition for the provincial Electric and Gas utility. I'm looking forward to using Pascal to implement our records & man scheduling system. At present Burroughs 1800, DEC PDP-11/70, Tandem, Univac 1100, Cyber 170 are all potential winners. I am particularly interested in Pascal on these machines." (*79/09/04*)

David L. Hamby, Combustion Engineering, INc., 1000 Prospect Hill Rd., Windsor, CT 06095: "Interests are real time process monitoring. Looking for process support software in a machine independent high level language." (*79/06/18*)

M. L. Harper, Oak Ridge National Labs, Bldg. 1505, Rm. 118, Oak Ridge, TN 37830: "I have pursued your references at JPL regarding a Pascal for ModComp minicomputers and the prospects look promising." (*79/06/26*)

David C. E. Holmes, P.O. Box 1708, Grafton, VA 23692: Teacher of micro-computer design, system design, and programming. owns 48K Z80 Altair 8800, CP/M, UCSD Pascal, and Ithica Intersystem Pascal/Z compiler. (*79/10/29*)

Mike Hughes, P.O. Box 393, Rapid City, SD 57709: "I am currently about three fourths of the way there on a business-oriented Pascal compiler for second-generation BCD machines. The implementation is for the RCA 301, but the problems are similar to the IBM 1401 and 1620, Burroughs B600, etc. I would be interested in getting in touch with anyone else having such Quixotic interests." (*no date*)

G. P. Janas, 4447 Buchanan, Warren, MI 48092: "I own an Apple][with two disk drives. I have on order, since September, the Apple Language Card and am awaiting same." (*79/10/18*)

Peter T. Jawbsen, Ceremain Microsystems, 759 Glen Canyon Rd., Santa Cruz, CA 95060: "I use both UCSD and OMSI Pascal." (*79/09/09*)

John W. Jensen, Jensen Farms, RR#1 Box 142, Everly, IA 51338: "I have been working on computer programs for a complete feedlot management system for about 4 years. The programs are written in RPG and run on an IBM System 34 which...I am losing access to.... I...am wiling to look at something in the $10-15000 range not counting software...(* here follows a description of hardware being considered *) Basic is the most popular language...but I'm not convinced that Basic is the best language to program in. Pascal has been called the software superstar. Yet it appears to me to be rather slow in being accepted. I have seen very little commerical software available (such as accounting packages, etc.)." (*79/10/01*)

Donald R. Kelley, 2451 Hingham Court, Woodbridge, VA 22192: "Just getting started using Pascal - have been working with assembly and BASIC." (*79/10/01*)

Wallace Kendall, 9002 Dunloggin Rd., Ellicott City, MD 21043: "I have an OSI Challenger III and have been trying for some time to get Pascal for it. Althought it has a Z80 chip (as well as a 6502 and a 6800) OSI apparently used a slightly different implementation, and the version used by most Z80 computers (I'm told) doesn't run on OSI. HOwever, I'm told that it will soon be ready either for the 6502 or the Z80 in OSI." (*79/05/07*)

Jack Laffe, 320 19th Ave. S., Minneapolis, MN 55454: "Re: machine dependent implementations: remove NCR 200 implementation that is listed in News #9/10 p. 105. This has been replaced by an NCR 8400 implementation and will be available February 1980. I will make more information available at that time." (*79/08/07*)

W. A. Lane, Canadian Tire Corporation, Limited, Box 770, Station K, Toronto, ONtario M4P 2V8: "We are a large retailing company in Canada with approximately 315 stores country wide. We are presently implementing "point of Sale" systems in these stores and are utilizing Datapoint, NCR and Amdahl computers. We also have several other machines including IBM system 34's, IV Phase and Basic mini's." (*79/08/22*)

James H. Lauterbach, Genesys Corporation, 223 Alexander Ave., Upper Montclair, NJ 07043: "Genesys Corporation...(*wishes*) to feature 'canned' applications programs which are easily customized...hence, our development system will probably be configured largely with C Basic and Pascal capability in mind--especially Pascal. Our quandary, at present, revolves arund the...relative merits of UCSD Pascal, the Per Brinch Hansen sequential version, the Intersystems Pascal/Z, the Alpha Micro version, the new 6809 Motorola version, the soon to be released Data General Micro NOva version, etc. etc. etc. Can you kindly bring some illumination to us?" (* no date *)

C. E. Leonard, 14008 S.E. Harrison, Portland, OR 97233: "I presently own an Exidy Sorcerer (Z80) with 32K and want to implement Pascal to go with my one year of Pascal studies at Portland Community College." (*79/08/31*)

Jerry LeVan, Eastern Kentucky University, Richmond, KY 40475: "I have extended Pascal-S with strings, scalars, graphics, execution profiler and many features useful in a teaching environment - runs under RSTS on a PDP-11/70." (*79/06/11*)

Robert C. Luckey, M.D., P.S., 1110 Gillmore Ave., Richland, WA 99352: "It is with distress that I read in the truly excellent issue 13 of your (*Andy's*) withdrawal from active lead position. You obviously have that combination of talent to co-ordinate a complex development such as that of a new high level computer language. None of the alternatives offered to the present arrangement at all compares with what we have now." (*79/03/26*)

Phong Thanh Ly, 6415 Prospect Terrace, Alexandria, VA 22310: "I am currently using Pascal on a PDP-11 and am going to have a Pascal compiler for the Honeywell Level-6 very soon." (*no date*)

Gregory A. Marks, Institute for Social Research, University of Michigan, SQR(A), MI 48106: "All I ever hear about UCSD Pascal is the good comments. Where can I get the opposite viewpoints; the problem in their extensions and implementation." (*79/06/29*)

Richard R. Martin, 634 Dallas Ave. #21, Grand Prairie, TX 75050: "I am running the UCSD Pascal on my Z80 system and am interested in keeping up with other implementations. My use for Pascal is in writing a CAI system with color graphics (RAMTEK). For a living, I manage a computer store." (*79/08/27*)

M. E. Markovitz, Culp & Tanner, Inc., 585 Manzanita Suite 6, Chico, CA 95926: "I am trying to build up a Pascal scientific library and would like to see if anyone else could lend me a hand. P.S. Does the user's group have such a scientific library?" (*79/07/23*)

Sakari M. Mattila, Lokkalantie 18 B 43, SF-00330 Helsinki 33, Finland: "I am a computer scientist at Technical Research Centre of Finland, EDP research division. We have University of Minnesota Pascal 6000 release 3 on CDC and some other on minis." (*79/07/07*)

Frank Monaco, 679 Lowell Drive, Marietta, GA 30060: "Keep up the good work." (*79/03/09*)

Jerry Moore, Dunn, Moore & Associates, 2935 E. Broadway, Suite 201, Tuscon, AZ 85716: "We are a systems house in Tucson working primarily with Perkin-Elmer (Interdata) and Alpha Microsystems minicomputers. We have a project slightly outside our normal sphere of influence, and...for which Pascal is most desirable. (*The project is*) a hydrologic model of complex irrigation systems for Saudi Arabian Naval base (* which *) must run on an IBM 3032 in Saudi Arabia. Development will have to be done on DEC system...unless I can find some IBM 370 time nearby. I would be very appreciative if you would consider my plight briefly and forward any suggestions." (*79/09/04*)

Hal Morris, Prindle and Patrick Architects:planners, 199 S. Fifth St., Columbus, OH 43215: "The company...is an architecture firm which has a PDP-11/34 running RT-11 and TSX. Our applications are Accounting, Word Processing, and some statistics and simulation.... My own impression is that C and Pascal are quite complementary, C being a better systems language, and Pascal being better for many, or even most applications." (*79/10/17*)

Gregory L. Nelson, Apt. 31, 2280 California St., Mountain View, CA 94040: "Have implemented Swedish Pascal V5 and NBS Pascal V1.4d (a preliminary version) under RSX-11M V3.1 on a PDP-11/70 system. Both Pascals lack operating system linkages sufficient to consider them for systems implementation." (*79/03/12*)

Neil Overton, Computer Systems and Services, Inc., Box 31407, Dallas, TX 75231: "I wanted an accounting package in Pascal or BASIC to be converted to run on a TI 990/2 for a large non-chain restaurant." (*79/09/05*)

Craig Payne, Enertec, 19 Jenkins Ave., Lansdale, PA 19446: "We are actively using Concurrent Pascal to write real time programs for the Z80. The language has been extended to allow the writing of device drivers directly in C.P.; the interpreter/kernel knows nothing about I/O." (*79/06/05*)

Raymond E. Penley, 3578F Kelly Circle, Bolling AFB, DC 20336: "Just purchased Pascal/Z from Ithaca Intersystems. This is a Z80 compiler that makes assembly code directly from the Pascal source. Will let you know more when I get it running. I don't have enough memory right now." (*79/09/24*)

Martin M. Peritsky, Bendix Corporation, P.O. Drawer 831, Lewisburg, WV 24901: "I am available for membership on standardization committees, etc. I am a member of IEEE and ISA. One of my specialties is compiler design." (*79/10/30*)

Stephen A. Pitts, 305 Jarman Dr., Midwest City, OK 73110: "I have ordered Apple Computer's Pascal system for my Apple][." (*79/08/24*)

Stephen M. Platt, 4060 Irving St., Philadelphia, PA 19104: "In my work (CS grad student U. of P.) people are starting to prefer Pascal to FORTRAN for reasons of portability(!) and ease of use. From my own view, it's a choice of hours debugging 100 lines of FORTRAN or not having to debug 700-1000 lines of Pascal...you get the idea. Keep up the good work." (*79/09/13*)

Michael S. Plesher, RDI Box 258, Hoewell, NJ 08525: "I am currently using the AAEC compiler on an IBM 370/168 (RCA, Cherry Hill NJ). They also have a Pascal P4 compiler." (*79/08/05*)

Hardy J. Pottinger, EE Dept., Univ. of Missouri Rolla, Rolla, MO 65401: "We are using University of Lancaster's implementation for Nova from Gamma Tech under RDOS and DOS. Like it a lot. We will be experimenting with microcomputer versions and concurrent Pascal during coming year." (*79/08/01*)

Fred W. Powell, P.O. Box 2543, Staunton, VA 22401: "I have been working primarily on a TI 990/10 computer which has a TI supported Pascal compiler. I expect to soon be using a TI 990/5 system which does not currently support the Pascal compiler. if TI does not change that problem soon, I intend to put the Pascal P compiler on that system. Thanks for your help and for the good job you are doing with PUG." (*79/10/08*) John Purvis, Sperry Univac Computer Systems, 55 City Centre Dr., Missisaugua, Ontario L5B 1M4: "I am a software instructor with Sperry Univac in Toronto. Our Mini Computer Operation is becoming involved with Pascal, so I am very interested in finding out what is happening with a Pascal user group." (*79/08/24*)

Frederick A. Putnam, Joseph R. Mares Asst. Prof., Dept. of Chemical Engineering, Massachusetts Institute of Technology, Cambrdige, MA 02139: "Here in the Chemical Engineering Department, we have a Data General Eclipse running (among other things) Gamma Technology's Pascal." (*79/10/17*)

Holly Robinson, Winthrop Publishers, Inc., 17 Dunster St., Cambridge, MA 02138: "We are about to publish two titles which will be of considerable interest to your PASCAL NEWS readership: PROGRAMMING FOR POETS: A GENTLE INTRODUCTION USING PASCAL, by Conway & Archer; and A PRIMER ON PASCAL by the same authors." (*79/10/03*)

Armando R. Rodriguez, P.O. Box 5771, Stanford, CA 94305: "I am in charge of the compilers for Pascal at LOTS, SAIL, GSB, SUMEX, and SCORE at Stanford, all of them DEC-10 or DEC-20. I am preparing a note on our improved version of the Hamburg compiler for DEC-10 and DEC-20." (*79/06/21*)

Wayne Rosing, Digital Equipment Corp., TW-C03, 1925 Andover St., Tewksbury MA 01876: "I was a 12/15/78 lost soul. I figured for $4/year you had gone out of business or you folks had been eaten by a FORTRAN compiler. (I'm on UCSD now but want to get a 32-bit Zurich version up on a 68000, demand paging off an 8 inch Winchester hard disk.)" (*79/08/20*)

Louis V. Ruffino, Federal Systems Division, IBM, 18100 Frederick Pike, Gaithersburg MD 20854: "Your pubs are excellent, but keep up the great work.
I look forward to PUG just like BYTE!" (*79/07/09*)

Carl Sandin, 314 Shadow Creek Dr., Seabrook, TX 77586: "I have a SOL-20, with North Star disks and Diablo printer. I'm trying to get started in North Star Pascal." (*79/08/06*)

Robert H. Scheer, CDP, Sheridan Oaks Cybernetics, 1915 Larkdale Dr., Glenview, IL 60025: "I have had some limited experience with Pascal on an Alpha Micro system and expect to start a project on a North Star Horizon microcomputer system before the year is over. I am also an instructor in computer science at Northwestern University's Division of Continuing Education in Chicago. I am investigating the possiblity of using Pascal as a means of teaching structured programming techniques." (*07/07/09*)

R. C. Shaw, The Grange, Spring Brank New Mills, Nr Stockport, Cheshire, SK12 4BH: "I would be interested in information on Pascal implementations on either Argus 700 or Modular One machines." (*07/09/13*)

Thomas W. Sidle, Technical Staff, Scientific Calculations, Inc., 4245-B Capitola, CA 95010: "We are interested in bringing up Pascal on VAX11/780, Prime 400 (and larger), and IBM 370/148 (and larger) computers." (*07/07/24*)

Connie Jo Sillin, Kansas City Southern Industries, Inc. 114 W. 11th St., Kansas City, MO 64105: "We at KCSI are interested in the Pascal programming language and the compiler for Pascal. We now have the IBM 370/158 and 3032 (OS-VS2) soon to be 3033 (MVS).

T. R. Simonson, G.M. Simonson & T.R. Simonson Consulting Engineers, 612 Howard Street, San Francisco, CA 94105: "I realize that PUG may have simply collapsed. I certainly hope not, for I have thoroughly enjoyed the contact. I believe you stated that some cross compilers exist for creating 8080 or Z80 machine code. If you know of one for CDC machines I would appreciate your jotting down the source." (*79/10/12*)

Lee L. C. Sorenson, 10226 Victoria Ave, Whittier, CA 90604: "I do not yet have a large enough system for Pascal, but I hope to learn from your group and to implement it in my system some day." (*79/06/07*)

T. J. Sullivan, 712 Rand Ave., Oakland, CA 94610: "I work with BART (*Bay Area Rapid Transit*) and am a neophyte to Pascal but am highly interested in all aspects of the language; particularly interested in programming for real time process control." (*79/06/07*)

Kevin Talbot, 3029 127th Place S.E., Bellevue, WA 98005: "The system I use is an HP3000 (Pascal P/3000 by Fraley, et. al.)" (*no date*)

Ron Tenny, President, G.W. Tenny Co. Inc., 3721 Scottsville Rd., Box A, Scottsville, NY 14546: "We are currently using a DEC 11/34 with 256KB memory, eight terminals, two printers, and dual 20MB drives in a business application environment. We want to implement Pascal under RSTS/E (CTS-500) and are looking for a good DBMS package to go with the Pascal code."

William W. Tunnicliffe, Bobst Graphic, INc., P.O. Box 462, Bohemia, NY 11716: "Thanks, volunteers!" (*79/08/20*)

Rex M. Venator, Major USA, 12451 Skipper Circle, Woodbridge, VA 22192: "While working on my Masters at Georgia Tech I became a Pascal 'fanatic' and since then my enthusiasm has not diminished. I attempt to follow all aspets of the language from the standardization efforts to Pascal's first descendant ADA in DOD. I would most certainly like to join your group and provide what assistance I can from an unofficial DOD perspective." (*79/05/16*)

Dick Wattson, 10 Dutton St. S., Manchester, NH 03104: "I surely would appreciate info on PDP-11 compilers (RT-11 compatible)." (*79/10/31*)

Anna Watson, 3705 Delwood Drive, Panama City, FL 32407: "Don't be discouraged, Andy. You're putting out a really interesting publication. I expect to use it as a reference tool later." (*79/08/12*)

Sydney S. Weinstein, CDP, CCP, 170 Centennial Road, Warminster, PA 18974: "I am now working for Fischer and Porter Company, and am developing data communications software for local networks for them. We use C as our main development language, but are also looking at Pascal especially as it develops for the PDP-11 and 8086 computers. Pascal is the basis of our new 'experimental' process control language." (*79/08/19*)

Tom Westhoff, Willmark A.V.T.I., Box 1097, Willmar, MN 56201: "Are there any Pascal implementations for Ohio Scientific Challenger II disk systems?" (*79/09/07*)

Rodney E. Willard, M.D., Loma Linda Medical Center Clinical Laboratory, Loma Linda, CA 92350: "I am trying to get a Z80 UCSD-CP/M system together and running." (*no date*)

R. S. Wood, 260 Trafalgar Lane, Aiken, SC 29801: "I'm a research analyst working for the DuPont Company at the Savannah River Laboratory. My interests in Pascal are both personal i.e., on a home micro and professional. The company is looking into the possiblity of using a Pascal based 'black-box' between our big main frames and any arbitrary microcomputer to make the micros look like all the other IBM-TSO terminals in the shop." (*79/07/03*)

Max Wunderlich, c/o Textronix, Inc., P.O. Box 500, Beaverton, OR 97077: "Both of us (*Max Wunderlich & Steve Jumonville*) are software engineers for Tektronix, Inc. We are presently using OMSI Pascal for production testing purposes on an LSI-11/2 with RT-11." (*no date*)

Richard Yensen, Ph.D., clinical Psychologist, 2403 Talbot Road, Baltimore, MD 21216: "I am running UCSD Pascal version I.5 on a Heathkit H-11 Computer with 32K words of 16 bit memory. The computer is a 16 bit machine." (*79/07/01*)

Fred Zeise, Data Systems Design, 3130 Coronado Drive, Santa Clara CA: "We are using ESI/OMSI Pascal and will be getting UCSD Pascal 1.5 soon." (*79/05/07*)

```
PPPPPP
P     P
P     P
PPPPPP
P
P
P  ASCAL IN THE NEWS
```

JOBS:

(* Note-these listings are intended primarily to show that there are indeed openings for Pascal programmers "out there". By the time you see these listings, the jobs may well be filled. *)

Control Data Corporation, Communications Systems Division, 3285 E. Carpenter Avenue, P.O. Box 4380-P, Anaheim, CA 92803: "Professional openings exist in the areas of data communications netowrk, message switching and front-end systems. Experienced candidates should be familiar in any of the following: Assembly/Pascal/Algol languages, Microprocessors, Real Time Systems, Communications protocols, test procedure development, test tool development." Contact Jess Holguin. (*Computerworld 79/09/24*)

Hewlett-Packard, West 120 Century Road, Paramus, NJ 07652: "We have opportunities both in Commercial and Scientific areas. Scientific experience is desired using FORTRAN, Assembler, BASIC, Pascal, data base, data communications with real-time operating systems. (*79/10/12*)

V.P. Personnel SS160, New York Times: "Minimum of 1 year experience. Programming experience with Pascal, PLM, P11, ALGOL, or FORTRAN" V.P. Personnel SS160 Times (*79/10/28*)

Perkin-Elmer Corporation, Main Avenue, NOrwalk, CT 06856: Looking for a micro-computer programmer whose responsibilities include "developing high level language (PL/1,Pascal) techniques to improve software development for micro-computers. (*79/10/28*)

MANUFACTURERS' ADVERTISEMENTS:

Apple Computer Co.,10260 Bandley Drive, Cupertino, CA 95014: Various advertisements for their version of UCSD Pascal

Columbia Data Products, Inc, 9050 Red Branch Road, Columbia, MD 21045: Advertising "a unique family of computer systems, the Commander series" which will run Pascal under CP/M. (* Computer Design, October 1979*)

Enertec, a company in Pennsylvania, has sent a flyer about their version of concurrent Pascal, which runs on the HP3000, and has an interpreter/kernel for a Z-80 Micro-computer. P-code for a given program is "about one-third the size of the P-code from Brinch-Hansen's concurrent Pascal compiler." On the Z-80, "execution speed at 4MHz is fast enough to handle 1200 baud terminals with all I/O to the IN, OUT level written in Concurrent Pascal. P-codes execute in 20 microseconds (push constant) to 500 microseconds (divide, context switch)

Pertec Computer Corp, Chatsworth, CA advertises a "Pascal Blaiser software development system, intended for systems and real-time applications programming," with 64K RAM, 1 megabyte of mass storage. The CPU directly executes Pascal; price is $5995 in single-unit quantities. (*Mini-Micro Systems October 1979*)

Rational Data Systems, 245 W 55th St., New York, NY 10019: has provided a Pascal that is "compatible with the entire (*Data General*) line - from Eclipse to microNova. All versions are source compatible and each can cross-compile for any of the other systems. The AOS version is priced at $3500." (*Computer Design, October 1979*)

Southwest Technical Products Corp., 219 W. Rhapsody, San Antonio, TX 78216 advertises the S/09 with MC6809 processor. "Both multiuser and multitasking/multiuser operating systems are available for the S/09. BASIC, Pascal, and an Assembler are immediately available." Cost with 128K bytes of RAM is $2995.

Sperry Univac Minicomputer OPerations, 2722 Michelson Dr., Irvine, CA 92713 has various advertisements for the Structured Programming System (SPS) running under their SUMMIT operating system which supports a Pascal compiler, debugger, program formatter, and concordance program. SPS also includes a text editor and document formatter.

Stirling/Bekdorf, 4407 Parkwood, San Antonio, TX 78218, advertises combination coding and CRT layout sheets to "speed software development and documentation for Pascal programmers". Two pads of 50 cost $26.85 plus $3.25 for handling.

Texas Instruments: Various advertisements for the DS990 Model which runs Pascal on a system that stores "up to 4,600,000 characters using double-sided, double-density diskette storage". Also advertisements in various places for their Microprocessor Pascal System with source editor, compiler, host debugger, configurator, native-code generator, and run-time support.

Three Rivers Computer Corp., 160 N. Craig St., Pittsburgh, PA. 15213: has a stand-alone system that can take up to 1 Megabyte of RAM, with interactive graphics (1024 lines on a 15-inch screen), and a speech output module. Mass storage is provided by 12 Megabyte Winchester disk drive with a 24 Megabyte disk option. "The unit contains a 16-bit processor that operates with P-Code, a high-level instruction language based on Pascal. The processor can reportedly execute in excess of one million P-Codes per second. The system's memory has a 32-bit segmented virtual addressing mechanism," and has 4K bytes of writable microstore as an option. (*Computerworld, 79/10/22*)

NEWSLETTERS & ARTICLES:

David A. Mundie has an article on the relative merits of Pascal vs. BASIC in Recreational Computing, Sept-Oct 1979. It concludes with "Most Pascal lovers are deeply committed to portability and standardization. It is not our fault that BASIC dialects have proliferated so wildly that there exists no standard BASIC to compare with Pascal."

Arthur Sale passes on a note from Computing, 1 November 1979, which mentions that the European Space Agency (ESA) will be using concurrent Pascal "to program ESA's latest venture into the simulation of satellite subsystems, the Multiple Processor Reconfigurable Simulator."

The Big Byte (University of Calgary) notes in its September 1979 issue that "the development of a Pascal compiler under Multics is near completion."

Early Warning Newsletter (University of Nebraska Computer Network) has a "new release of Stanford Pascal. This version is a considerable improvement over previous versions. For the most part, changes to the system are enhancements and will not affect Pascal programs that ran under the previous version." A change has been made to nested comments, giving a compiler option to make constructs such as (* x:=y (* comment *) *) legal or produce an error as the user desires. (* 79/09/13*)

Log On (Massey University Computer Centre), notes that "We are to implement a Pascal compiler" for a newly-acquired IBM Series/1 minicomputer. In usage statistics for the B6700, Pascal comes in second place with 10% of usage (981 accesses) during June 1979. (*July 1979*)

ICSA Newsletter (Rice University, Houston TX), tells "Pascal users don't despair. Although Pascal is currently not available at ICSA, we hope to remedy the situation soon. Plans are underway to install Pascal 8000 this fall." (*79/09/17*)

BOOKS ABOUT PASCAL

Alagic, S. and Arbib, M. S., *The Design of Well-structured and Correct Programs*, Springer-Verlag, 1978, 292 pages.

Bowles, K. L., *Microcomputer Problem Solving Using Pascal*, Springer-Verlag, 1977, 563 pages.

Brinch Hansen, P., *The Architecture of Concurrent Programs*, Prentice-Hall, 1977.

Coleman, D., *A Structured Programming Approach to Data*, MacMillan Press, 1978, 222 pages.

Conway, R. W., Gries, D. and Zimmerman, E. C., *A Primer on Pascal*, Winthrop Publishers Inc., 1976, 433 pages.

Findlay, B. and Watt, D., *PASCAL: An Introduction to Methodical Programming*, Computer Science Press (UK Edition by Pitman International) 1978.

Grogono, P., *Programming in Pascal*, Addison-Wesley, 1978, 359 pages. *Note: Those persons using the first printing of this text may obtain a list of corrections from: Barry Cornelius, Dept. of Computer Studies, University of Hull, Hull, HU6 7RX, England.*

Hartmann, A. C., *A Concurrent Pascal Compiler for Minicomputers*, Sprinter-Verlag Lecture Notes in Computer Science, No. 50, 1977.

Jensen, K. and Wirth, N., *Pascal User Manual and Report*, Springer-Verlag Lecture Notes in Computer Science, No. 18, 2nd Edition, 1976, 167 pages.

Kieburtz, R. B., *Structured Programming and Problem-Solving with Pascal*, Prentice-Hall Inc., 1978, 365 pages.

Rohl, J. S. and Barrett, H. J., *Programming via Pascal*, Cambridge University Press, in press.

Schneider, G. M., Weingart, S. W., and Perlman, D. M., *An Introduction to Programming and Problem Solving with Pascal*, Wiley and Sons, 1978, 394 pages.

Webster, C. A. G., *Introduction to Pascal*, Heyden, 1976, 129 pages.

Welsh, J. and Elder, J., *Introduction to Pascal*, Prentice-Hall Inc., in press.

Wilson, I. R. and Addyman, A. M., *A Practical Introduction to Pascal*, Springer-Verlag, 1978, 148 pages.

Wirth, N., *Systematic Programming: An Introduction*, Prentice-Hall, 1973, 169 pages.

Wirth, N., *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976, 366 pages.

Alagic, S.; Arbib, M. A. "The Design of Well-Structured and Correct Programs," Springer-Verlag, New York, 1978.

> The major goal of this book is to present the techniques of top-down program design and verification of program correctness hand-in-hand. It thus aims to give readers a new way of looking at algorithms and their design, synthesizing ten years of research in the process. It provides many examples of program and proof development with the aid of a formal and informal treatment of Hoare's method of invariants....
>
> The secondary goal of this book is to teach the reader how to use the programming language Pascal....
>
> From the Preface

This reviewer is a Pascal production programmer and this review is presented in light of that background. While many production programmers, not familiar with the Pascal language, may find this book to be somewhat difficult at first reading, it is well worth the trouble for the insights that it provides. The production programmer, considering the purchase of this book, should have a well read copy of Jensen and Wirth [1] handy. This book's advantage is that it can raise the programming abilities of its careful readers. The chapters and the topics chosen for inclusion are:

Chapter	Topic
1	Introducing Top-Down Design
2	Basic Compositions of Actions and Their Proof Rules
3	Data Types
4	Developing Programs with Proofs of Correctness
5	Procedures and Functions
6	Recursion
7	Programming with and without Gotos

Chapter 2 contains an excellent introduction to logical formulas; Chapter 3 contains an excellent primer on set theory (expanded later in Chapter 4). A bibliography, glossary and subject index are included as are two appendices: the syntax of Pascal and a complete renumeration of Pascal statement Proof Rules. Typography is clean and uncluttered with extremely few typographical errors.

I have only two complaints regarding this book. The first, an annoyance, is the excessive use of reference numbers appended to examples. The authors also begin reference renumbering at the section level rather than at the chapter level. This causes unnecessary difficulties to the reader who, ignoring the section number, provided at the top of the odd-numbered pages, thumbs back to find a referenced example (in one case, the reference is to an example in a preceding section, therefore requiring a little detective work to determine exactly which example should be reviewed!) I have found myself completely baffled by an 'obviously erroneous' backward reference, only to realize, after some consternation, that I had passed back into an earlier section!

The second, and perhaps more significant, complaint deals with the formatting of and symbols used in Pascal program examples. The indentation scheme is inconsistent. Thus, on page 89, we find:

```
while ¬ eof(f) do
        begin S := S + f↑;
              get (f)
        end
```

while on the very next page (90), we find

```
for i := 1 to numstud do
begin gr := grade [i,j] ;
      if gr ≠ 0 then totgrade := totgrade + gr
      else numgrades := numgrades - 1
end
```

In the first example, it is clear that the compound statement is within the scope, and therefore control, of the while; in the second it is not at all apparent that the compound statement is under the control of the for. Although this inconsistency may be a symptom of a 'gremlin typesetter', it should be corrected in future editions. A less disconcerting problem with the typesetting of Pascal programs is the use of the non-Pascal symbols '∧', '∨', '¬' and '≠'. Since they are not a part of the language, they should be replaced by and, or, not and '<>', respectively, in all program fragments (they are acceptable within the proof comments, since they have a logical meaning).

This text has been used in at least one graduate level course and so contains material of interest to the more erudite Pascal programmer. Even though the going may be rough at times, I stongly recommend this book to anyone seriously interested in programming languages, and especially to Pascal programmers.

G. G. Gustafson, San Diego CA

Reference

[1] Jensen, K. and Wirth, N. "PASCAL - User Manual and Report," Second Edition (Corrected Printing), Springer-Verlag, New York, 1978.

ARTICLES ABOUT PASCAL

Addyman, A. M., et al., "A Draft Description of Pascal," Software - Practice and Experience, Vol. 9, 381-424, (1979).

Atkinson, L. V.,"Pascal Scalars as State Indicators," Software - Practice and Experience, Vol. 9, 427-431, (1979).

Ball, M. S., "Pascal 1100: "An Implementation of the Pascal Language for Univac 1100 Series Computers," NTIS: AD-A059 861/5WC, (1 Jul 78).

Barron, D., "On Programming Style, and Pascal," Computer Bulletin, 2,2, (Sep 79).

Bate, R. R. and D. S. Johnson, "Putting Pascal to Work," Electronics, (7 Jun 79).

Bishop, J. M., "On Publication Pascal," Software - Practice and Experience, Vol. 9, 711-717, (1979).

Bishop, J. M., "Implementing Strings in Pascal," Software - Practice and Experience, Vol. 9, 779-788, (1979).

Bonyun, D. A. and Holt, R. C., "Euclid Compiler for PDP-11," NTIS: AD-A061 402/4WC, (Apr 78).

Bonyun, D. A. and Holt, R. C., "Euclid Compiler for PDP-11," NTIS: AD-A061 406/5WC, (Oct 78).

Brinch Hansen, P. and Hayden, C., "Microcomputer Comparison," Software - Practice and Experience, Vol. 9, 211-217, (1979).

Clark, R. G., "Interactive Input in Pascal," ACM SIGPLAN Notices, (Feb 79).

Crider, J. E., "Structured Formatting of Pascal Programs," ACM SIGPLAN Notices, (Nov 78).

Davis, H., "The Pascal Notebook," Interface Age, Chapter 1, (Jun 79).

Fletcher, D., Glass, R. L., Shillington, K., and Conrad, M., "Pascal Power," Datamation, (Jul 79).

Forsyth, C. H. and Howard, R. J., "Compilation and Pascal on the New Microprocessors," Byte, (Aug 78).

Gracida, J. C. and Stilwell, R. R., "NPS-Pascal. A Partial Implementation of Pascal Language for a Microprocessor-based Computer System," NTIS: AD-A061 040/2WC, (Jun 78).

Graef, N., Kretschmar, H., Loehr, K., Morawetz, B., "How to Design and Implement Small Time-sharing Systems Using Concurrent Pascal," Software - Practice and Experience, Vol. 9, 17-24, (1979).

Graham, S. L., Berkeley, U. C., Haley, C. B., and Joy W. N., "Practical LR Error Recovery," ACM SIGPLAN Notices, (Aug 79).

Grogono, P., "On Layout, Identifiers and Semicolons in Pascal Programs," ACM SIGPLAN Notices, (Apr 79).

Gustafson, G. G., "Some Practical Experiences Formatting Pascal Programs," ACM SIGPLAN Notices, (Sep 79).

Hansen, G. J., Shoults, G. A., and Cointment, J. D., "Construction of a Transportable, Multi-pass Compiler for Extended Pascal," ACM SIGPLAN Notices, (Aug 79).

Heimbigner, D., "Writing Device Drivers in Concurrent Pascal," ACM SIGOPS, (Nov 78).

Holdsworth, D., "Pascal on Modestly-configured Microprocessor Systems," IUCC Bulletin, 1, 1, (1979).

Holt, R. C., and Wortman, D. B., "A Model for Implementing Euclid Modules and Type Templates," ACM SIGPLAN Notices, (Aug 79).

Joslin, D. A., "A Case for Acquiring Pascal," Software - Practice and Experience, Vol. 9, 691-692, (1979).

LeBlanc, R. J., "Extensions to Pascal for Separate Compilation," ACM SIGPLAN Notices, (Sep 78).

LeBlanc, R. J., and Fischer, C., "On Implementing Separate Compilation in Block-Structured Languages," ACM SIGPLAN Notices, (Aug 79).

Luckham, D. C., and Suzuki, N., "Verification of Array, Record, and Pointer Operations in Pascal," ACM Transactions on Programming Languages and Systems, Vol. 1, 2, (Oct 79).

Marlin, C. D., "A Heap-based Implementation of the Programming Language Pascal," Software - Practice and Experience, Vol. 9, 101-119, (1979).

Narayana, K. T., Prasad, V. R., and Joseph, M., "Some Aspects of Concurrent Programming in CCNPASCAL," Software - Practice and Experience, Vol. 9, 749-770, (1979).

Natarajan, N., and Kisinha, M., "Language Issues in the Implementation of a Kernel," Software - Practice and Experience, Vol. 9, 771-778, (1979).

Nelson, P. A., "A Comparison of Pascal Intermediate Languages," ACM SIGPLAN Notices, (Aug 79).

Nievergelt, J., et al., "XS-0: A Self-explanatory School Computer," Dr. Dobb's Journal of Computer Calisthenics and Orthodontia, No. 36, (Jun/Jul 79).

Parsons, R. G., "UCSD Pascal to CP/M File Transfer Program," Dr. Dobb's Journal of Computer Calisthenics and Orthodontia, Box E. Menlo Park, CA 94025, No. 37, (Aug 79).

Perkins, D. R., and Sites, R. L., "Machine-independent Pascal Code Optimization," ACM SIGPLAN Notices, (Aug 79).

Powell, M. S., "Experience of Transporting and Using the SOLO Operating System," Software - Practice and Experience, Vol. 9, 561-569, (1979).

Pugh, J. and Simpson, D.,"Pascal Errors - Empirical Evidence," Computer Bulletin, (Mar 79).

Ravenel, B. W., "Toward a Pascal Standard," IEEE Computer, (Apr 79).

Rudmik, A. and Lee, E. S., "Compiler Design for Efficient Code Generation and Program Optimization," ACM SIGPLAN Notices, (Aug 79).

Sale, A., "SCOPE and PASCAL," ACM SIGPLAN Notices, (Sep 79).

Sale, A. H. J., "Strings and the Sequence Abstraction in Pascal," Software - Practice and Experience, Vol. 9, 671-683, (1979).

Schauer, H., "MICROPASCAL - A Portable Language Processor for Microprogramming Education," Euromicro Journal, 5, 89-92, (1979).

Schneider, G. M., "Pascal: An Overview," IEEE Computer, (Apr 79).

Shimasaki, M., et al., "A Pascal Program Analysis System and Profile of Pascal Compilers," Proceedings of the Twelfth Hawaii International Conference on System Sciences, (ED.) Fairley, R. E., (1979).

Silberschatz, A., "On the Safety of the IO Primitive in Concurrent Pascal," Computer Journal, Vol. 22, No. 2, (May 79).

Sites, R. L. and Perkins, D. R., "Universal P-Code Definition," NTIS: PB-292 082/5WC, (Jan 79).

Sites, R. L., "Machine-independent Register Allocation," ACM SIGPLAN Notices, (Aug 79).

Smith, G. and Anderson, R., "LSI-11 Writable Control Store Enhancements to U. C. S. D. Pascal," NTIS: UCIO-18046, (Oct 78).

Tanenbaum, A. S., "A Comparison of Pascal and ALGOL 68," Computer Journal, Vol. 21, No. 4, (Nov 78).

Tanenbaum, A. S., "Implications of Structured Programming for Machine Architecture," Communications of the ACM, (Mar 78).

Wallace, B., "More on Interactive Input in Pascal," ACM SIGPLAN Notices, (Sep 79).

Watt, D. A., "An Extended Attribute Grammar for Pascal," ACM SIGPLAN Notices.

Wickman, K., "Pascal is a Natural," IEEE Spectrum, (Mar 79).

Wiggers, R. and Van De Riet, R. P., "Practice and Experience with BASIS: An Interactive Programming System for Introductory Courses in Informatics," Software - Practice and Experience, Vol 9., 463-476, (1979).

Wirth, N., "MODULA-2," ETH Zurich, Institut für Informatik, No. 27, (Dec 78).

Wirth, N., "Reflections About Computer Science," Univ. of York (England) Dept. of Computer Science, Report No. 19, (Jul 78).

Wirth, N., "A Collection of Pascal Programs," ETH Zurich, Institut für Informatik, No. 33, (Jul 79).

☆ ☆ ☆ ☆ ☆ ☆ ☆ ☆

UCSD Workshop Proceedings

The Proceedings of the July 1978 UCSD Workshop on Pascal Extensions (see Pascal News #13, pages 12..15) are now available for $25 from:

Institute for Information Systems
Mail Code C-021
University of California, San Diego
La Jolla, CA 93093
USA

Payment must accompany all orders.

{ Several persons involved with the Workshop expressed to me their unhappiness with the Proceedings. Because of this, I asked Ruth Higgins, who served on the Editorial Board, to provide some background information. Ruth graciously agreed to do so, and the following note is the result. }
-Jim Miner

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

Comments on the Proceedings of the UCSD Workshop on System Programming Extensions to the Pascal language.

The Proceedings of the UCSD Workshop on System Programming Extensions to the Pascal Language are now available. I would like to provide some information for the benefit of those who did not attend the workshop but will obtain a copy of the proceedings.

Near the end of the second week of the Workshop, it became clear that we would not be able to approve the wording of a final document within the time frame of the Workshop. And yet, since the proceedings would be purported to represent consensus of about 50 industry representatives, it was important that they be accurate. To that end, the Workshop participants appointed an Editorial Board whose function was to compile a draft of the proceedings for UCSD to distribute to Workshop attendees for comment with respect to accuracy, review those comments, attempt to edit the draft to reflect the comments and prepare a final version. Preparation and distribution of copies was provided by the Information Sciences Institute, UCSD.

The Editorial Board met in August, 1978, to prepare the draft. It was distributed to Workshop members with the phrase "Not for distribution" on each page. The comment period was to last until the end of October. The next date when most of the Editorial Board could meet was January 11, 1979. At that time, we went through each section of the proceedings and tried to incorporate comments as fairly as possible. We then wrote instructions to Gillian Ackland, the UCSD person who was doing the actual editing and distribution of the document. We also wrote a cover letter to accompany the proceedings. Copies of both of these are enclosed.

In late April or early May, I received a phone call from Gillian. She said she had had a very busy winter quarter and had not been able to do anything at all on the proceedings. However, in the Spring, she had gone on with the work but had a few questions. Instructions 1 through 5 (see enclosed) were OK, but why didn't the Editorial Board members want their names included except in the Workshop attendees list? I told her that we had discussed this at length and agreed that we did not want our names to lend credibility or be misconstrued as endorsement of the poor technical quality of the document.

She had another question regarding Section G (Proposed Experiments) on the subsection on Type Secure External Compilation. This section had sparked several, carefully written, long letters disputing the accuracy of what claimed to be a representation of the part on which there had been agreement. The Board could find no way to treat these fairly except to instruct Gillian to include the letters also in that section. For some reason, Ken Bowles and Terry Miller did not want to do that. Instead, they left the section as it was in the first draft and added, as an editorial comment, the sentence "The accuracy of this representation has been disputed." She asked me if that was all right. I said that the Board had considered that approach but felt it would be educationally important to include all of the disagreement to show how pervasive the dispute was. Anything less would be misleading and, therefore, unfair to the workshop participants. Gillian suggested that they rewrite the section, incorporating the comments as best they could. I told her that the rewritten section would have to be approved by, at least, those who had disputed the first version. It seemed to me that the simplest, fairest, and most professionally honest way to handle it was to make the whole technical controversy available to the readers. In addition, it would help to demonstrate how complicated the issue of external compilation really is.

When one receives a copy of the proceedings one can see that the cover letter is not included; the words "not for distribution" do not appear as per the Board's instructions; and the subsection on Type Secure External Compilation does not include any of the related technical controversy. Finally, a final copy was not sent to the Editorial Board Chairman as requested in 8 (see enclosed). I was told that the matter was handled in such a way in the interest of time, that the whole thing had dragged on far too long and any further delay was not justified compared to the desirability of getting it distributed. It is not clear to me how the Board's instructions could have added noticeable delay.

Sincerely,

Ruth M. Higgins

Ruth M. Higgins

Enc.(2)

Jan, '79

Dear Gillian:

Many thanks for getting your new version of Sections B thru F to us. There was some concern about how certain comments had been handled. Having the updated version allowed us to check.

We have decided that, on the basis of responses from reviewers, the proceedings do not merit publication. However, the Workshop participants deserve an accurate report. Therefore, enclosed are the required corrections.

Regarding overall format,

1. Replace Section A with the enclosed;
2. Edit Sections B through G as shown. Although you did not send us your copy of G, the Board edited a copy from the first draft to our complete satisfaction;
3. Delete Section H, Section I, and Appendix X;
4. Insert page numbers in the Table of Contents;
5. The list of participants should be in alphabetical order by name of individual accompanied by affiliation, omitting addresses and phone numbers.
6. The members of the Editorial Board do not wish to have their names appear anywhere except among those of Workshop participants.
7. Since the Board feels that these proceedings do not merit wide distribution (even though persons requesting individual copies should receive them at cost), the phrase NOT FOR DISTRIBUTION will remain on each page.
8. Before printing, mail a final copy to Bruce Ravenel. He will ascertain that editing instructions were understood correctly.

Thank you again for your tremendous efforts. We appreciate the work you have done so far. Good Luck in this semester!

The Editorial Board

To: The Workshop Participants

From: The Editorial Board

Subject: The Enclosed Proceedings

Date: January 11, 1979

This is the final version of the Proceedings to the UCSD Workshop on System Programming Extensions to the Pascal language.

In light of review responses received, the Editorial Board has decided that the quality of the contents of this document merits distribution to the Workshop participants only. It does not warrant publication. However, as prescribed in the general resolutions (Section B), copies will be sent to a few others and will be available at reproduction and mailing costs to any who request individual copies. Recipients of this document are requested to restrain from distributing it further.

The production of these Proceedings reflect the combined efforts of many people. In particular, Gillian Ackland has performed an outstanding, Herculean effort of document preparation and distribution under the guidance of Terry Miller and Ken Bowles. We wish to thank them on behalf of the Workshop participants.

A Report on Pascal Activities at the
San Diego 1979 Fall DECUS U.S. Symposium

Bill Heidebrecht
TRW DSSG
One Space Park
Redondo Beach, CA 90278

The 1979 Fall Digital Equipment Computer Users Society (DECUS) U.S. Symposium was held in San Diego, California on December 10-13. Approximately 600 of the 2500 people who preregistered indicated an interest in Pascal. The DECUS Pascal SIG, chaired by Dr. John R. Barr of the University of Montana, has now grown to over 2000 members.

In the Pascal Implementation Workshop, John Barr, Brian Nelson and I spoke briefly about the implementation of NBS Pascal under RSX, RT-11, RSTS and VAX/VMS systems. Gerry Pelletier of Transport Canada spoke about his work in implementing a self compiling version of Torstendahl's "Swedish" Pascal (V5.3) under RSX-11M.

In the Pascal Standards Report, Leslie Klein (DEC) and Barry Smith (Oregon Software) reported on the current status of the ISO draft standard and progress within the X3J9-IEEE Joint Pascal Committee. Barry gave a detailed discussion on conformant array parameters and answered a number of good questions from the audience. The quality of questions asked showed the increasing level of sophistication of Pascal users in the DEC world.

John Barr gave a presentation of his work on implementing NBS Pascal on LSI 11's running RT-11. The compiler is completely selfsupporting now on such systems, and can compile itself on a 28K word machine using the RT-11 SJ monitor. It takes approximately 10 minutes to compile the compiler on an LSI-11 using floppy disks (about 700 lines/minute). The compiler is not yet a full implementation of Standard Pascal, but we (the Pascal SIG) are working on it.

William Donner and James Forster of TMI Systems gave interesting presentations on the implementation of a financial message switch for EFT using a Pascal Multi-Process Subsystem (PMPS-11), which they also implemented. They added concurrency facilities (processes, monitors and semaphores) to OMSI Pascal strictly by adding to the runtime, without extending the language. Fed up with MACRO, FORTRAN and RATFOR, they considered using C, PL/I and Pascal as their implementation language. They chose Pascal for its reliability, efficiency and good structure. 99% of their system is written in Pascal.

Isaac Nassi of Digital Equipment gave two overview presentations on Ada, which were very well attended. The audience seemed somewhat overwhelmed by the complexity of the language.

During the Pascal SIG Business Meeting a variety of topics was discussed. For example, Leslie Klein gave an update on DEC's VAX Pascal compiler. The compiler has undergone field testing since June 79 at 15 sites, and should be ready for shipment to customers very soon (approx. December 79). Although it is not a highly optimizing compiler, the test sites were largely enthusiastic about it. One of the test site users reported moving a large program from CDC Pascal to the VAX with only 3 changes to the program required. DEC should start receiving some user feedback on the compiler by the next DECUS Symposium.

Reid Brown of Digital spoke about the positive influence the Pascal SIG has had on Digital with respect to Pascal.

Roy Touzeau (Pascal SIG Newsletter Editor) and John Barr also spoke on a number of subjects concerning the SIG. Due to DECUS's new funding structure, each SIG may soon have to charge a small annual subscription fee for its newsletter.

I spoke briefly about the status of the DECUS Pascal SIG library. The Fall 79 Pascal SIG library contains two versions of Seved Torstendahl's "Swedish" Pascal: version 6, which contains some new symbolic debugging facilities, and the version modified by Gerry Pelletier to enable it to compile itself on a PDP11. There are also versions of NBS Pascal for RSX, RSTS and RT-11 systems, as well as a number of other utilities. PN readers who are interested in the Pascal SIG library should consult recent editions of the DECUS Pascal SIG Newsletter for more details.

The next DECUS U.S. Symposium will be held in Chicago on April 22-25, 1980, and will again feature a number of interesting Pascal sessions.

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

ISO/TC 97/SC 5 N **553**

ISO INTERNATIONAL ORGANIZATION FOR STANDARDIZATION
ORGANISATION INTERNATIONALE DE NORMALISATION

From: UNIPREA
VIA MONTEVECCHIO, 29
10128 - TORINO

Telephone, 53 17 12
Telegrams: UNISO - TORINO

ISO/TC 97/SC 5

PROGRAMMING LANGUAGES

Secretariat ANSI (U.S.A.)

REPORT ON ADA

Ada is a programming language being produced by the U.S. Department of Defense in cooperation with several foreign and international organizations. The project has spanned five years and is unique for its openness in all phases and the resultant international contributions.

The first phase was an evolution of requirements from the users by an itterative process which produced five versions, increasingly refined. These documents were widly circulated and major input was received from individuals outside the U.S., from the International Purdue Workshop such especially its LTPL-E committee, and from experts of SC 5/WG 1. Major support has been contributed by the CEC and by the goverments of the U.K. and Germany. We believe that this requirements phase was very valuable in settling many of the questions that normally arise much later in the development process, when they are much more difficult to deal with. It might be said that, in the best procedure for major projects, we are proceeding thorough definitive requirements, followed by firm design, before coding.

After evaluation of several dozen existing languages against these requirements, a new design was initiated. On the basis of an international request for proposal, four contractors were chosen to produce competive prototypes. All started from Pascal, although there is no intent that the resulting language be closely related to Pascal, since their requirements were much different. The initial designs from these four contractors were reviewed by several hundred experts worldwide and a decision was made to continue refinement of two of the designs. A year later, these two designs were reviewed, again with international participation. The single design selected was that produced by Cii Honeywell-Bull. That design, and a document giving rationale for design decisions, are contained in N-499 and have been distributed as the June 1979 issue of SIGPLAN Notices. A preface from the Secretary of Defense requests international public comment.

For any that do no have this document, a microfiche is available this meeting.

Ada is a modern powerful computer programming language. It has real-time features and has been under consideration by WG 1 for that reason. It is however targeted to a much wider audience.

Ada promotes modularity for the production of large systems, strong data typing for reliable, even provable, programming, etc. A rigorous definition will allow control of the language to make possible wide portability. It is our intent that there be no subset or superset compilers and that a validation facility be used to assure compliance.

Our economic analyses show that even more benefit may be attributed to the commonality resulting from exactly compatible systems than that would be attributed to the technical improvements postulated from introduction of Ada.

Even greater benefits may accrue from the wide availability if tools a development environment, debugging systems, applications specific packages, etc. We term this the "environment" of Ada. It is expected that the availability of this environment to those who have compliant compilers will be an incentive for such compliance.

A fundamental question is why does the DoD want to get involved with national and international standardization. Ada is being volved in a single place and does not have the normal standards problem of rationalization of divergent definitions and implementations. Is not the DoD's control sufficient?

It may well be that the DoD has sufficient control internally and with its contractors. This control may be sufficient to carry over to much of U.S. industry. We are not confident that this will be sufficient to cover small business, academic, and foreign industry. We do, however, feel very strongly about the benefits of commonality, specifically those benefits to the DoD of universal commonality, the ability to pick up programs generated elsewhere, transfer of technology, availability of compilers generated elsewhere, and most significantly the increacend availability of other sources on which we can draw for hardware and software contractors, increacing competition.

For the advantages this will provide, the DoD is prepared to relinguish some control to the proper authorities, the matter is certainly up for negotiation. Ada Control Board will be established to maintain and interpret the standard. It seems reasonable to have representatives on this group from any nation having a significant committment to the language. Consider that group as the sponsoring body, presently the U.S. DoD with representatives of U.K., France and Germany.

It has certainly been true that the design of ADA, and the entire project leading up to it, has been an international effort, as I believe has been evidenced here today. It would be a shame if this opportunity to assure, from the beginning, a worldwide single definition was missed.
In light of the resolution 6 intent, we consider that we are now in a phase of simultaneous comment from local, national, and international bodies. This was the purpose of the WG 1 Resolution and the SC 5 circulation of the documents (N 499, N 504, N 505).
Several hundred comments have already been received and processed. The results of these comments and further studies will result in a final design document in May 1980 (with perhaps an early draft in January 1980). At that time we will have a Military Standard, and, one expects, a US Government Standard. I belive that at that time, with your cooperation, we will have done the processing appropriate in order for SC 5 to recommend Ada for international standardization.

UNL
The University of Nebraska-Lincoln

Department of Computer Science
Ferguson Hall
Telephone (402) 472-2402
Lincoln, Nebraska 68588

Pascal User's Group, c/o Andy Mickel
University Computer Center: 227 EX
208 SE Union Street
University of Minnesota
Minneapolis, MN 55455

Dear Andy,

Enclosed is an article for the Pascal News that should be of interest to your readers. It describes some observations on error message frequency, persistence, and apparent student reaction in an introductory Pascal class for Computer Science majors and advocates the development of better error diagnostics particularly for novice programmers.

Sincerely,

David W. Embley

David W. Embley
Assistant Professor

A STUDY OF SYNTAX ERRORS ENCOUNTERED BY BEGINNING PASCAL PROGRAMMERS

Kirk Baird
David W. Embley
Department of Computer Science
University of Nebraska - Lincoln
Lincoln, NE 68588

1. Introduction

In the 1978-1979 school year, the Computer Science Department at the University of Nebraska - Lincoln replaced FORTRAN with PASCAL as the introductory language for Computer Science majors. Since PASCAL was known to only a handful of upperclassmen and professors, it was anticipated that beginning students would encounter difficulty finding assistance with errors in their programs. The traditional sources of assistance, other than the teaching assistant or professor (e.g. the debug consultant, fraternity files, or the dorm-floor Comp. Sci. genius) would not be as helpful as before. In this situation, increased dependence on the compiler generated error messages was inevitable; and even though PASCAL is designed for instructional use, its error diagnostics are unfortunately not composed so that the beginning student can readily understand them.

Anticipating this difficulty, we decided to observe all first semester student programs submitted for execution and note error message frequency, error persistence, and apparent student reaction and catalogue actual causes for each error. The results of these observations were to serve as a basis for improving PASCAL error messages or at least to provide material for a reference document for beginning PASCAL programmers.

2. Data Collection

The students observed were Computer Science majors taking CS 155, Introduction to Computer Programming, using PASCAL. These students ran their PASCAL programs on an IBM 370/148 (later upgraded to a 158) using the September 1977 version of a PASCAL compiler developed at Stanford University.

A special JCL package was developed for use in data collection. Each time a student ran a program, the output, including in-line error messages, was routed to disk. If the program compiled without syntax errors, it was allowed to execute, and the output was also sent to disk. A copy of all of the temporary disk output including program listing and program output was placed in a permanent file and finally routed to the printer and given to the student as if it were undisturbed. The permanent file was occasionally reblocked and copied to tape.

The data collected in this manner eventually came to almost six million bytes of storage. Elementary pattern matching techniques were used to locate and tabulate the occurrences of syntax errors in this data. The results of this tabulation appear in Appendix I.

On occasion, listings of random portions of the data were printed, and the syntax errors, their cause, and their persistence were analyzed by hand and cataloged. Later in the semester, printouts of unsuccessful runs were collected by the professor and turned over for analysis and cataloging. The results of this tabulation are reported in Appendix II.

3. Observations

Three general observations can be made from the data: 1) beginning students interpret error messages too literally, 2) differences between standard PASCAL as described in the text (Kieburtz, 78) and the version implemented confuse students, and 3) certain error messages seem to be particularly ambiguous or misleading.

3.1 Literal Interpretation

Given little else, the beginning student is likely to depend unwittingly on the compiler generated error messages, at first taking them too literally. In the Stanford compiler as implemented at UNL, an error arrow points to a particular column of a line of code and is followed immediately by a list of error message numbers. The premise is made that the arrow points to the exact position of the error described by the error messages associated with the error numbers. In fact, the error arrow never points to the exact position of the error. Most often, it is positioned just past the error, usually pointing at the following keyword or identifier.

More than once a student forgot to put a semicolon at the end of the PROGRAM line and found the error arrow pointing to the character following the succeeding keyword, VAR, giving the message "SEMICOLON EXPECTED". The student would run the program a second time with a semicolon after the keyword (i.e. VAR;), and the compiler would respond with an error arrow pointing to the semicolon and the message "SEMICOLON EXPECTED", among others.

Other students inadvertently put a semicolon where a comma belongs in a WRITELN parameter list. The resulting error was ") EXPECTED" with the error arrow positioned near the semicolon. Subsequent runs showed students putting right parentheses before, after, and in place of the semicolon.

3.2 A Non-Standard Version

The second problem is the difference between the standard version of PASCAL and the one implemented at UNL. Since some characters were not available, the compiler expected standard substitutions such as left-parentheses-vertical-bar for left-square-bracket and the at-sign for up-arrow. These obvious distinctions caused relatively few problems.

Some other differences, however, were more detrimental. For example, in the September 1977 version of the Stanford compiler, the standard identifier MAXINT was not implemented, nor was PAGE, and WRITELN and its counterparts had to be followed by parentheses in contrast to the syntax diagrams. Several students faithfully adhered to the syntax diagrams and appropriately omitted the parentheses only to find their code blemished with unwarranted syntax errors. The subsequent July 1978 version resolved the problems with PAGE and WRITELN but disallowed SET OF CHAR. Hence students copying segments of programs from their text with such syntactically legal expressions as `CH IN (| 'A'..'Z' |)` or `N >= SQRT(MAXINT)` would get syntax errors.

3.3 Ambiguity

The third problem is the ambiguity of the error message itself. There are a handful of often occurring ambiguous error messages including "ILLEGAL SYMBOL" and "ERROR IN VARIABLE" and less often occurring messages such as "SEMICOLON EXPECTED" and "TYPE CONFLICT OF OPERANDS". In fact, "ILLEGAL SYMBOL" and ERROR IN VARIABLE" accounted for almost forty percent of all error messages observed.

One of the most often committed blunders exemplifies the novices reaction to these ambiguous messages. Students would precede an ELSE with a semicolon; the resulting error message, "ILLEGAL SYMBOL", pointed at the blank following the ELSE. Students replaced this blank with almost anything, including another THEN, another semicolon, a BEGIN, or a new line.

The reason ambiguous error messages hold such a majority of the total is twofold: 1) the very fact that the error message is unclear causes the student to repeat it, sometimes with changes, and at times with the innocent hope that it will go away, and 2) many error messages have more than one cause and are unclear because the message has to be general enough to cover all cases.

4. What can be done?

Ideally, the compiler should be modified, with the beginning student in mind, to give more appropriate error messages. This modification should involve more than mere cosmetic changes to the error messages. Most likely, additional messages are needed, and a finer distinction among possible causes should be incorporated particularly for ambiguous and high frequency error messages.

Not having developed the compiler ourselves, we were not in a position to make these intricate alterations. We were, however, in a position to alter the error message table so that an error message would include a listing of the most prevalent potential sources of the error. Although this option was at our disposal, we rejected it for a number of reasons. No beginning student could remain calm at seeing a hard-worked-on, twenty-line PASCAL program intermingled with two hundred lines of error messages. Moreover, there are certain to be sources of errors that have not been cataloged; a given student assignment might generate a particular error message a thousand times even though it never appeared during the semester observed. In addition, because Stanford is regularly updating its compiler, such alterations would soon be made obsolete. For example, when a literal character string spanned two source lines on the September 1977 version, the error message generated was "IMPLEMENTATION RESTRICTION". In subsequent versions, the error is "STRING CONSTANT CANNOT EXCEED SOURCE LINE".

In view of these difficulties, it was thought best to provide a supplementary handout that could be updated from time to time. This handout (Baird, 79) provides a list of the most frequently encountered errors and their typical causes. Another advantage of a handout over a cosmetic alteration of the syntax error table is that additional documentation and helpful suggestions can also be included. In addition to syntax errors, this handout documents differences between the UNL Stanford compiler and standard PASCAL, describes runtime errors and what to do about them, lists compiler options, and shows and explains a sample program listing.

We encourage PASCAL implementors to make the effort to provide better error messages particularly for novice programmers. We would be interested to hear of such projects in progress and would eventually like to obtain a compiler with error messages that are more palatable to the beginner.

References

1. Kieburtz, R. B., Structured Programming and Problem Solving with PASCAL, Prentice Hall, 1978.

2. Baird, K., "Stanford PASCAL at UNL", Department of Computer Science, University of Nebraska - Lincoln, 1979.

APPENDIX I

These errors were tabulated from students running PASCAL as an introductory programming language, using the Stanford PASCAL compiler. The actual error message is listed in order of decreasing occurrence. Errors of insignificant occurrence are omitted.

	ERROR	PERCENT OCCURRENCE
6	: ILLEGAL SYMBOL	27.0
104	: IDENTIFIER IS NOT DECLARED	18.2
59	: ERROR IN VARIABLE	11.4
13	: "END" EXPECTED	04.5
58	: ERROR IN FACTOR	04.3
*****	: END OF FILE ENCOUNTERED	04.1
398	: IMPLEMENTATION RESTRICTION	03.6
134	: ILLEGAL TYPE OF OPERAND(S)	02.7
51	: ":=" EXPECTED	02.5
4	: ")" EXPECTED	02.4
101	: IDENTIFIER DECLARED TWICE	02.1
5	: ":" EXPECTED	01.6
129	: TYPE CONFLICT OF OPERANDS	01.6
10	: ERROR IN TYPE	01.5
103	: IDENTIFIER IS NOT OF APPROPIATE CLASS(sic)	01.5
18	: ERROR IN DECLARATION PART	01.4
14	: ";" EXPECTED	01.3
125	: ERROR IN TYPE OF STANDARD FUNCTION PARAMETER	01.0
2	: IDENTIFIER EXPECTED	00.8
144	: ILLEGAL TYPE OF EXPRESSION	00.7
21	: "*" EXPECTED	00.6
52	: "THEN" EXPECTED	00.6
116	: ERROR IN TYPE OF STANDARD PROCEDURE PARAMETER	00.5
17	: "BEGIN" EXPECTED	00.4
53	: "UNTIL" EXPECTED	00.4
54	: "DO" EXPECTED	00.4
124	: F-FORMAT IS FOR REAL TYPE ONLY	00.4
9	: "(" EXPECTED	00.3
140	: TYPE OF VARIABLE IS NOT A RECORD	00.3
50	: ERROR IN CONSTANT	00.2
126	: NUMBER OF PARAMETERS DOES NOT AGREE WITH DECLARATION	00.2
145	: TYPE CONFLICT	00.2
8	: "OF" EXPECTED	00.1
16	: "=" EXPECTED	00.1
20	: "," EXPECTED	00.1
55	: "TO" OR "DOWNTO" EXPECTED	00.1
102	: LOW BOUND EXCEEDS HIGHBOUND	00.1
106	: NUMBER EXPECTED	00.1
107	: INCOMPATIBLE SUBRANGE TYPES	00.1
139	: INDEX TYPE IS NOT COMPATIBLE WITH DECLARATION	00.1
142	: ILLEGAL PARAMETER SUBSTITUTION	00.1
143	: ILLEGAL TYPE OF LOOP CONTROL VARIABLE	00.1
150	: ASSIGNMENT TO STANDARD FUNCTION IS NOT ALLOWED	00.1
167	: UNDECLARED LABEL	00.1
201	: ERROR IN REAL CONSTANT : DIGIT EXPECTED	00.1
255	: TOO MANY ERRORS IN THIS SOURCE LINE	00.1

Appendix II

The following error messages were found in the programs of beginning PASCAL students and were catalogued as to what caused them. Only the more recurrent causes are listed; the obvious causes are not listed (e.g. error 14, ";" EXPECTED, does not list missing semicolon as a cause).

2: IDENTIFIER EXPECTED

a) extra comma in list
b) used TYPE as a variable name
c) missing quote in character literal
d) previous error in declaration
e) used zero instead of O in identifier

4: ")" EXPECTED

a) => used instead of >=

5: ":" EXPECTED

(note: in Stanford PASCAL, the colon is a viable substitute for ..)

a) tried to use FILE as a variable name
b) CASE without END
c) TO used instead of ..

6: ILLEGAL SYMBOL

a) previous statement missing a semicolon
b) semicolon precedes ELSE
c) misspelled keyword
d) => instead of >=
e) missing quote in character literal
f) missing (in comment
g) = used instead of :=
h) extra END
i) DO used instead of BEGIN
j) TO used instead of ..
k) = used instead of : for RECORD within RECORD
l) END missing on CASE statement
m) comma missing in list
n) spaces within an identifier
o) comma or colon used instead of a semicolon

8: "OF" EXPECTED

a) tried to use FILE as a variable name
b) identifier declared twice

10: ERROR IN TYPE

a) tried to use TYPE as a variable name
b) colon used instead of equal sign

13: "END" EXPECTED

a) forgot END for RECORD
b) used TYPE as a variable name within record

14: ";" EXPECTED

(note: this error only occurs within the declaration part
semicolons missing within the block are flagged with
error 6: ILLEGAL SYMBOL)

a) illegal characters within PROGRAM identifier
b) forgot END for RECORD
c) tried to redefine TYPE within a RECORD

16: "=" EXPECTED

a) colon used to instead of equal sign
b) tried to use TYPE as a variable within a RECORD

18: ERROR IN DECLARATION PART

a) VARIABLES used instead of VAR

19: ERROR IN FIELD LIST

a) forgot END for RECORD

50: ERROR IN CONSTANT

a) ... used instead of ..
b) TO used instead of ..
c) variable list used as an array index

51: ":=" EXPECTED

a) = used instead of :=
b) misspelled name of procedure identifier

58: ERROR IN FACTOR

a) => used instead of >=
b) literal character used without quotes
c) real fraction constant used without leading zero

59: ERROR IN VARIABLE

a) missing quote
b) missing semicolon
c) missing comma in list
d) misspelled procedure identifier
e) := used instead of = in expression
f) misspelled AND
g) illegal charactrer in identifier

101: IDENTIFIER DECLARED TWICE

a) identifier used once as an element in a user defined
datatype and once as a simple variable

102: LOWBOUND EXCEEDS HIGHBOUND

a) TO used instead of ..

103: IDENTIFIER IS NOT OF APPROPRIATE CLASS

a) semicolon missing before WRITE
b) previous error in declaration
c) no END for CASE statement
d) missing quote for literal string

104: IDENTIFIER IS NOT DECLARED

a) misspelled identifier
b) misspelled keyword
c) missing quote in character literal
d) imbedded blanks within an indentifier

116: ERROR IN TYPE OF STANDARD PROCEDURE PARAMETER

a) tried to read a user defined datatype qualified record
identifier broken between source lines

125: ERROR IN TYPE OF STANDARD FUNCTION PARAMETER

a) passed integer to TRUNC

129: TYPE CONFLICT OF OPERANDS

a) integer assigned a real result
b) misspelled identifier
c) / used instead of DIV
d) literal character string not the same size as
ARRAY OF CHAR it is assigned to

134: ILLEGAL TYPE OF OPERAND(S)

a) => used instead of >=
b) previous error in declaration

136: SET ELEMENT MUST BE SCALAR OR SUBRANGE

a) set written inside square brackets
e.g. X : SET OF BOOLEAN; .... (| X |)

138: TYPE OF VARIABLE IS NOT AN ARRAY

a) = used instead of := when assigning an array

139: INDEX TYPE IS NOT COMPATIBLE WITH DECLARATION

a) previous error in declaration

140: TYPE OF VARIABLE IS NOT A RECORD

a) previous error in declaration

143: ILLEGAL TYPE OF LOOP CONTROL VARIABLE

a) previous error in declaration

144: ILLEGAL TYPE OF EXPRESSION

a) := used instead of =

145: TYPE CONFLICT

a) previous error in declaration

147: LABEL TYPE INCOMPATIBLE WITH SELECTING EXPRESSION

a) no END for CASE statement

152: NO SUCH FIELD IN THIS RECORD

a) misspelled field
b) previous error in declaration

156: MULTIPLY DEFINED CASE LABEL

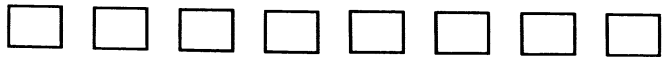
a) no END for CASE statement
b) missing quote within CASE statement
c) ELSE preceded by semicolon in CASE statement

255: TOO MANY ERRORS IN THIS SOURCE LINE #

(note: the compiler only lists the first nine syntax errors of a source line)

398: IMPLEMENTATION RESTRICTION

a) WRITELN (a record)
b) literal character string > 64 characters
c) SETs OF CHAR are disallowed on the compiler



Applications

AYLI - As You Like It

Production programming in Pascal requires a number of source code manipulation tools. With them appropriate application specific syntactic sugar and common multi-program procedure and data structure definitions can be managed. Doug Comer's MAP is such a program.

Tram's complex arithmetic routines and Judy Bishop's/Arthur Sale's string routines are examples of typical library source utilities. Barry Smith also sent in a small string package. Take your pick. After all, with Pascal you can have it AYLI.

CORRECTIONS

A Class of Easily ... - Pascal News #15
in example #3 change
"Y = Ø" to "Y = 9"

Applications

S-5 "ID2ID" (See PN 15, September 1979, page 31.)

Jim Miner spotted two typos in the published version of ID2ID. He also provided code to improve error processing by handling unclosed strings correctly as well as an unexpected EOF inside comments. - Andy Mickel

Correct typographical errors:

Replace line 172 by:

```
                    if P2^.Bal = HigherRight then P1^.Bal := HigherLeft
```

Replace line 314 by:

```
    ImportantChars := LettersAndDigits + ['(', '{', ''''];
```

Improve error processing:

Replace line 3 by:

```
*         James F. Miner     79/06/01, 79/09/30.
```

Insert after line 275:

```
    label
      1 { TO ESCAPE EOF INSIDE OF A COMMENT };
```

Replace lines 338 and 339 by:

```
                                'in source program.')
                else begin Write(Target, Source^);   Get(Source) end
```

Insert after line 345:

```
                    Write(Target, Source^);   Get(Source);
```

Delete line 347.

Replace lines 350, 351, and 352 by:

```
                        if EOLn(Source) then
                          begin WriteLn(Target);   ReadLn(Source);
                            if EOF(Source) then goto 1 { EXIT SCAN }
                          end
                        else begin Write(Target, Source^);   Get(Source)   end
```

Replace lines 362, 363, and 364 by:

```
                  if EOLn(Source) then
                    begin WriteLn(Target);   ReadLn(Source);
                      if EOF(Source) then goto 1 { EXIT SCAN }
                    end
                  else begin Write(Target, Source^);   Get(Source) end
```

Replace line 372 by:

```
    end;
  1: { COME FROM EOF INSIDE OF COMMENT }
```

****

USER MANUAL - REFERENCER

Version S-02.01, 1979 December 17

1. INTRODUCTION

The Referencer program is a software tool intended to assist programmers in finding their way around Pascal program listings of non-trivial size. In keeping with a basic philosophy that software tools should have distinct and clear purposes (as indeed most craftsmen desire), the function of Referencer has been defined as providing a compact summary of procedure-headings in a program, and a table of calls made by each procedure. It thus provides information on the first-order procedural interfaces.

The products of Referencer may serve also as an adjunct to a full cross-reference, because in presenting less information Referencer produces a more convenient summary. Additionally, for those people who wish to change the syntax of Pascal to require repetition of a procedure-heading at the occurrence of the block of a forward-declared procedure, it will serve as a reminder that language changes are not the only answer to every problem.

2. USE OF REFERENCER

Version S-02.01 of Referencer, the distribution version, has no options to be set. It reads from the input file, expecting to find a complete Pascal program on this textfile. Although the results with syntactically incorrect programs are not guaranteed, Referencer is not sensitive to most flaws. It cares about procedure, function, and program headings, and about proper matching of begins and cases with ends in the statement-parts.

The two tables are produced on the file output. Referencer does not try to format the headings to fit them into a device-line width; it leaves them pretty much as they were entered into the program, except for indentation alignment. The first table thus benefits from a wide print-line. The second table has a constant in the program which controls its width, and the distributed version requires 132 characters of print positions.

Thus, use of Referencer involves simply executing it, with the attachment of the input file to some program text, and the direction of the output file to some suitable printing device.

3. LEXICAL STRUCTURE TABLE

The first table (see Appendix) displays the lexical structure and the procedure headings. (The term procedure means procedure, function, or program in this documentation unless otherwise stated.) As the program is read, each heading is printed out with the line-numbers of the lines in which it occurs. The text is indented on the first line so as to display the

lexical nesting. Subsequent lines are adjusted left or right so as to maintain their relative position with respect to this 'mother' line. On rare occasions it may not be possible to achieve this adjustment if there are insufficient leading spaces to delete on the dependent lines, and then the display will suffer.

In this context, the 'procedure heading' is taken to mean all the text between and including the opening reserved word of the heading, and the semicolon that separates it from the text that follows. What will be printed is everything contained on the lines that contain this heading. While this definition of procedure heading is not the one in the draft Pascal Standard, it is a pragmatic convenience to consider it thus rather than as the syntactic construct.

The prime use of this table is in understanding programs; it documents the interfaces to each procedure, their lexical nesting, and where the headings are located.

4. THE CALL-STRUCTURE TABLE

The second table is produced after the program has been scanned completely, and is the result of examining the internal data. For each procedure listed in alphabetical order, the table holds:

- The line-number of the line on which its heading starts.
- Unless it was external or formal (and had no corresponding block), the line-number of the begin that starts the statement-part (the body).
- In the Notes column, the characters 'ext' are printed if the procedure has an external body (declared with a directive other than forward), and the characters 'fml' are printed if it is a formal procedural or functional parameter. If a number appears, the procedure has been declared forward and this is the line-number of the line where the block of the procedure begins; the second part of the two-part declaration.
- A list of all user-declared procedures immediately called by this procedure. In other words, their call is contained in the statement-part. The list is in order of occurrence in the text; a procedure is not listed more than once if it is called more often.

This table may be useful in finding the components of a procedure as they are squashed into different places in the listing by the flattening effects of syntax. It may also be useful in seeing the inter-dependencies of the procedures of the program.

5. LIMITATIONS

As mentioned before, the behaviour of Referencer when presented with incorrect Pascal programs is not guaranteed. However, it has been the intention that it be fairly robust, and there are not too many flaws that will cause it to fail. The most critical features, and therefore those likely to cause failure if not correct are the general structure of the procedure heading (reserved word followed by name with optional parameter-list with balanced parentheses followed by semicolon and either reserved word or directive), and the correct matching of end with each begin or case in each statement-part (since this information is used to detect the end of a procedure).

If an error is explicitly detected, and Referencer has very few explicit error checks and minimal error-recovery, a message is printed out that looks like this:

```
FATAL ERROR - No identifier after prog/proc/func - AT FOLLOWING LINE
procedure ( t : TransactionType );
```

The line of text printed is where the program was when it got into trouble; like all diagnoses this does not guarantee that the correct parentage is ascribed to the error. Processing may continue despite the fatal error for a while, but the second table will not be produced.

Referencer is believed to accept the full Pascal language, as described in the draft proposal submitted to ISO, and to process it correctly.

6. PORTABILITY

It is believed that Referencer uses only Standard Pascal features according to the draft proposal submitted to ISO.

It should be relatively easy to transfer it to other Pascal processors. It does not use packing, except for pseudo-strings of characters. Neither does it use dispose, though a possible usage is marked in the program. The small amount of data stored does not warrant their use if it might imperil portability. It requires the use of small sets of at least set of 0..15, and a set of char. Those who have not a set of char available can fairly easily program around it, and complain to their Pascal suppliers. The names are stored internally in a canonic letter-case (lower-case in Version S-02.01), with a set indicating those to be transformed on output. This strategy should enable users to modify it to run even on CDC's 6+6 bit lower-case system, and on one-case systems. The program implements the Pascal Standard's attitude towards letter-case.

7. SYSTEM NOTES AND MODIFICATIONS

7.1 PARAMETERIZED CONSTANTS

The heading of the program contains information on altering:

- The significance limit of identifiers (currently 16 characters). This should not be reduced below 10 as it will be difficult to distinguish identifiers and reserved words.

- The difference between upper-case and lower-case letters. EBCDIC users will probably need to change only this single constant.

- The line width for table 2, which automatically affects the number of columns of called procedure names. The distributed version has this set at 132, which allows 5 columns of 16-character names across the page. Setting it to 54, which allows a single column, is an useful variation.

- The number of indentation spaces per level.

7.2 INTERNAL STRUCTURES

Procedure information is held in an 'Entry' record, each of which is linked into two binary trees by alphabetical order of name (ignoring letter-case). Each 'Entry' record contains a linked list of 'UsageCell's which point to procedures called from that procedure. There is also a lexical stack display, composed of 'StackCell's. Similarly, these point to the currently nested procedures during the first phase of processing. Each stach cell also contains a root pointer which holds a "scope-tree" which contains all the names declared at this level. A single "super-tree" contains all the procedure names. The scope-trees are traversed during searching for names, and the supertree is used to produce the final table.

The final tables are capable of further interpretation which has not been done here in the interests of simplicity of the resulting software tool. For example, recursivity may be deduced from the data, and small modifications would allow the keeping of call-frequency counts.

As mentioned earlier, each name is separated into a case-independent component and a solely-case component for storage. The identifiers are reconstructed at the time of display. In the case where not all occurrences of an identifier have the same visual representation, Referencer will thus still recognize them as the same, and will use the first occurrence as the display form. Referencer could easily check the identity of such forms, but any error messages would spoil the tables and it has not been done in line with the philosophy that each tool has a particular purpose. General-purpose tools are often such compromises that they are successful at none of their tasks...

7.3 EFFICIENCY

As might be expected, Referencer spends most of its time in NextCh, NextToken, ReadIdent, IgnoreComment and FindNode. As a guide, the following information was collected while Referencer processed its own text. The counts under the "Statements" column are the maximum statement counts for any statement within the procedure body. All counts have been rounded and depend to some extent on the use of spaces and tabs in the source file.

Procedure	Calls	Statements
NextCh	30800	30800
NextToken	2600	8700
ReadIdent	1600	9000
FindNode	3800	4500
IgnoreComment	102	13500
...	...	...

The space usage of Referencer is very small, except perhaps for the program itself.

On Berkeley Pascal running under UNIX on a PDP-11/34, processing Referencer by itself requires about 96 seconds of processor time. This is about 10.6 lines per second. The code occupies about 9,000 bytes of storage. Berkeley Pascal is an interpretive system intended for student users, and is therefore rather slow in comparison with compilers with native code generation.

8. ERROR REPORTING

If any errors in processing Standard Pascal programs are detected, please write to the author at the following address with the exact details. Problems with processing incorrect or non-Standard programs are not interesting.

Prof A.H.J.Sale
Department of Information Science
University of Tasmania
Box 252C, G.P.O. Hobart
Tasmania 7001

Any experiences with the portability of this tool are also welcomed. A Technical Report on its design and structure is in preparation.

9. HISTORY

This program grew out of the proper haunts of good ideas (the coffee-room) and several discussions of what one would like from such a tool. A.J.Currie, at the University of Southampton, produced the first prototype program of 231 lines. Based on this experience and the problems in accepting the full Pascal language, A.H.J.Sale (on leave from the University of Tasmania) wrote the current version of just over 1000 lines. The resulting program is now about 20% slower than the prototype, but it is believed to be a more modifiable and a correct tool.

The current program was written in 4 days. It does not fit into any integrated system of software tools but has been designed with the basic view that software tools should be plentiful, correct, portable, flexible, and single-purpose. All attributes are equally important.

Arthur Sale

```
0001 program Referencer(input,output);
0002 {-------------------------------------------------------------------------}
0003 |                                                                         |
0004 |                  PASCAL PROCEDURAL CROSS-REFERENCER                     |
0005 |                                                                         |
0006 |         (c) Copyright 1979 A.H.J.Sale, Southampton, England.            |
0007 |                                                                         |
0008 |   DEVELOPMENT                                                           |
0009 |   This program is a software tool developed from a prototype by         |
0010 |   A.J.Currie at the University of Southampton, England. The proto-      |
0011 |   type of 231 lines of source text was used firstly as a basis for      |
0012 |   extensions, and then rewritten to assure correctness by               |
0013 |   A.H.J.Sale, on leave from the University of Tasmania and then         |
0014 |   also at the University of Southampton. The current version was        |
0015 |   stabilized at 1979 December 4; the development time being es-         |
0016 |   timated at 4 man-days from prototype to production.                   |
0017 |                                                                         |
0018 |   PURPOSE                                                               |
0019 |   The program reads Pascal source programs and produces two tables      |
0020 |   as output. These tables are procedural documentation and cross-       |
0021 |   references. One documents all procedure or function headings in       |
0022 |   a format that illustrates lexical nesting. The other tables           |
0023 |   gives the locations of heading, block, and body for each pro-         |
0024 |   cedure and function, and what procedures and functions it immedi-     |
0025 |   ately calls.                                                          |
0026 |                                                                         |
0027 |   There is a User Manual for this program; if it has not been pro-      |
0028 |   vided with your installation write to:                                |
0029 |            Department of Information Science                            |
0030 |            University of Tasmania                                       |
0031 |            P.O.Box 252C, G.P.O. Hobart                                  |
0032 |            Tasmania  7001                                               |
0033 |   and ask for the Technical Report on Referencer, if it is still        |
0034 |   available. The program is written to be portable and is believed      |
0035 |   to be in Standard Pascal.                                             |
0036 |                                                                         |
0037 |   Permission is granted to copy this program, store it in a comput-     |
0038 |   er system, and distribute it, provided that this header comment       |
0039 |   is retained in all copies.                                            |
0040 |                                                                         |
0041 |-------------------------------------------------------------------------}
0042
0043 {-------------------------------------------------------------------------}
0044 |                                                                         |
0045 |                          PROGRAM ASSERTIONS                             |
0046 |                                                                         |
0047 |   Pre-Assertion P1:                                                     |
0048 |         "The file input contains a representation of a correct          |
0049 |         Standard Pascal program, in the ISO Reference form."            |
0050 |                                                                         |
0051 |   Post-assertion P2:                                                    |
0052 |         P1 and "the file output contains a representation of the        |
0053 |         two tables described above, which correctly describe facts      |
0054 |         about the program."                                             |
0055 |                                                                         |
0056 |-------------------------------------------------------------------------}
0057
0058 const
0059     { This constant is the number of significant characters kept in
0060     the identifier entries.  It can readily be changed.  It is not
0061     advised that it be reduced below 10 (reserved words get to 9). }
0062     SigCharLimit = 16;
0063
0064     { This must always be (SigCharLimit - 1).  It is used simply to
0065     reduce the set range to have a lower bound of 0, not 1. }
0066     SetLimit = 15;
0067
0068     { This constant is used to convert upper-case letters to lower-case
0069     and vice-versa.  It should be equal to ord('a') - ord('A'). }
0070     UCLCdisplacement = 32;
0071
0072     { This constant determines the size of the input line buffer.
0073     The maximum acceptable input line is one smaller because a sentinel
0074     space is appended to every line. }
0075     LineLimit = 200;
0076
0077     { This constant determines the maximum width of the printing of the
0078     second cross-reference table.  The program deduces how many names
0079     will fit on a line. }
0080     LineWidth = 132;
0081
0082     { This determines the indentation of the lex-levels. }
0083     Indentation = 4;
0084
0085     { These constants are used for the sketchy syntax analysis.
0086     They are collected here so that their lengths may be altered if
0087     SigCharLimit is altered. }
0088     Sprogram      = 'program        ';
0089     Sprocedure    = 'procedure      ';
0090     Sfunction     = 'function       ';
0091     Slabel        = 'label          ';
0092     Sconst        = 'const          ';
0093     Stype         = 'type           ';
0094     Svar          = 'var            ';
0095     Sbegin        = 'begin          ';
0096     Scase         = 'case           ';
0097     Send          = 'end            ';
0098     Sforward      = 'forward        ';
0099     Spaces        = '               ';
0100
0101 type
0102     Natural       = 0..maxint;
0103     Positive      = 1..maxint;
0104
0105     SixChars      = packed array[1..6] of char;
0106
0107     SigCharRange  = 1..SigCharLimit;
0108     SetRange      = 0..SetLimit;
0109
0110     PseudoString  = packed array [SigCharRange] of char;
0111     StringCases   = set of SetRange;
0112
0113     LineSize      = 1..LineLimit;
0114     LineIndex     = 0..LineLimit;
0115
0116     SetOfChar     = set of char;
0117
0118     ProcKind      = (FwdHalf,AllFwd,Shortform,Formal,Outside,NotProc);
0119
0120     PtrToEntry    = ↑ Entry;
0121
0122     ListOfUsages  = ↑ UsageCell;
0123
0124     PtrToStackCell = ↑ StackCell;
0125
0126     TokenType     = (OtherSy,NameSy,LParenSy,RParenSy,ColonSy,
0127                      SemiColSy,PeriodSy,AssignSy,SubRangeSy);
0128
0129     { This type represents a procedure or function identifier found
```

```
0130     during processing of a program.  The fields are used as follows:
0131       - procname & caseset = representation of name
0132       - linenumber         = where heading starts
0133       - startofbody        = where begin of statement-part starts
0134       - forwardblock       = where forward-declared block starts
0135       - status             = kind or status of name
0136       - left,right         = subtrees of the scope-level tree
0137       - before, after      = subtrees of the supertree
0138       - calls              = a list of the procedures this calls
0139       - localtree          = the scope tree for the interior
0140     }
0141     Entry =
0142           record
0143               procname     : PseudoString;
0144               caseset      : StringCases;
0145               linenumber   : Natural;
0146               startofbody  : Natural;
0147               left,right   : PtrToEntry;
0148               before,after : PtrToEntry;
0149                calls       : ListOfUsages;
0150               localtree    : PtrToEntry;
0151               case status: ProcKind of
0152                   FwdHalf,Shortform,Formal,Outside,NotProc:
0153                       ();
0154                   AllFwd:
0155                       ( forwardblock: Natural )
0156           end;
0157
0158     { This type records an instance of an activation of a procedure or
0159     function.  The next pointers maintain an alphabetically ordered
0160     list; the what pointer points to the name of the activated code. }
0161     UsageCell =
0162           record
0163               what: PtrToEntry;
0164               next: ListOfUsages
0165           end;
0166
0167     { This type is used to construct a stack which holds the current
0168     lexical level information. }
0169     StackCell =
0170          record
0171              current: PtrToEntry;
0172              scopetree: PtrToEntry;
0173              substack: PtrToStackCell
0174          end;
0175
0176 var
0177     lineno       : Natural;
0178     chno         : LineIndex;
0179     total        : LineIndex;
0180     depth        : Natural;
0181     level        : -1..maxint;
0182     pretty       : Natural;
0183
0184     { These are used to align the lines of a heading. }
0185     adjustment   : (First,Other);
0186     movement     : integer;
0187
0188     {These are true, respectively, if line-buffers need to be
0189     printed before disposal, and if any errors have occurred. }
0190     printflag    : Boolean;
0191     errorflag    : Boolean;
0192
0193     ch           : char;
0194
0195     token        : TokenType;
0196
0197     symbol       : PseudoString;
0198     symbolcase   : StringCases;
0199
0200     savesymbol   : PseudoString;
0201
0202     line         : array[LineSize] of char;
0203
0204     superroot    : PtrToEntry;
0205
0206     stack        : PtrToStackCell;
0207
0208     { The remaining variables are pseudo-constants. }
0209     alphabet     : SetOfChar;
0210     alphanums    : SetOfChar;
0211     uppercase    : SetOfChar;
0212     digits       : SetOfChar;
0213     usefulchars  : SetOfChar;
0214
0215     namesperline : Positive;
0216
0217     procedure PrintLine;
0218     var
0219          i : LineSize;
0220     begin
0221          write(output, lineno:5, '   ');
0222           i := 1;
0223          { Is this the first time in a run or not? }
0224          if adjustment = First then begin
0225              { Ignore any leading spaces there happen to be. }
0226              while (i < total) and (line[i] = ' ') do
0227                   i := succ(i);
0228              { Compute the adjustment needed for other lines. }
0229              movement := (level * Indentation) - (i - 1);
0230              adjustment := Other;
0231              { Insert any necessary indentation }
0232              if level > 0 then
0233                   write(output, ' ': (level*Indentation));
0234          end else begin
0235              { It wasn't the first time, so try to adjust this
0236              line to align with its mother. }
0237              if movement > 0 then begin
0238                  write(output, ' ':movement)
0239              end else if movement < 0 then begin
0240                  while (i < total) and (line[i] = ' ') and
0241                      (i <= -movement) do begin
0242                      i := succ(i)
0243                  end
0244              end
0245          end;
0246          { Write out the line. }
0247           while i < total do begin
0248               write(output, line[i]);
0249               i := succ(i)
0250           end;
0251           writeln(output)
0252     end; { PrintLine }
0253
0254     procedure Error(e: Positive);
0255     { This procedure is the error message repository. }
0256     begin
0257          errorflag := true;
0258          write(output, 'FATAL ERROR - ');
```

```
0259          case e of
0260           1: write(output, 'No "program" word');
0261           2: write(output, 'No identifier after prog/proc/func');
0262           3: write(output, 'Token after heading unexpected');
0263           4: write(output, 'Lost ".", check begin/case/ends');
0264           5: write(output, 'Same name, but not forward-declared')
0265          end;
0266          { We shall print the offending line too. }
0267          writeln(output, ' - AT FOLLOWING LINE');
0268          adjustment := First;
0269          PrintLine
0270      end; { Error }
0271
0272      procedure NextCh;
0273      begin
0274            if chno = total then begin
0275                if printflag then
0276                    PrintLine;
0277                total := 0;
0278                while not eoln(input) do begin
0279                   total := succ(total);
0280                    read(input, line[total])
0281                end;
0282               total := succ(total);
0283               line[total] := ' ';
0284               readln(input);
0285                lineno := lineno + 1;
0286                chno := 1;
0287               ch := line[1]
0288            end else begin
0289                chno := succ(chno);
0290                ch := line[chno]
0291            end
0292      end; { NextCh }
0293
0294      procedure Push(newscope: PtrToEntry);
0295      var
0296            newlevel: PtrToStackCell;
0297      begin
0298            new(newlevel);
0299            newlevel↑.current := newscope;
0300           newlevel↑.scopetree := nil;
0301            newlevel↑.substack := stack;
0302            stack := newlevel;
0303            level := level + 1
0304      end; { Push }
0305
0306      procedure Pop;
0307      var
0308            oldcell: PtrToStackCell;
0309      begin
0310           stack↑.current↑.localtree := stack↑.scopetree;
0311            oldcell := stack;
0312            stack := oldcell↑.substack;
0313            { *** dispose(oldcell); *** }
0314            level := level - 1
0315      end; { Pop }
0316
0317      procedure FindNode(var match   : Boolean;
0318                         var follow  : PtrToEntry;
0319                             thisnode: PtrToEntry);
0320      begin
0321            match := false;
0322            while (thisnode <> nil) and not match do begin
0323                follow := thisnode;
0324                if savesymbol < thisnode↑.procname then
0325                    thisnode := thisnode↑.left
0326                else if savesymbol > thisnode↑.procname then
0327                    thisnode := thisnode↑.right
0328                else
0329                    match := true
0330            end
0331      end; { FindNode }
0332
0333      function MakeEntry (mainprog: Boolean;
0334                          proc    : Boolean): PtrToEntry;
0335      { The first parameter is true if the name in symbol is the
0336      program identifier, which has no scope.  The second parameter
0337      is true if the name in symbol is that of a procedure or function.
0338      The result returned is the identification of the relevant record. }
0339      var
0340            newentry, node: PtrToEntry;
0341            located: Boolean;
0342
0343           procedure PutToSuperTree(newnode: PtrToEntry);
0344           { This procedure takes the entry that has been created by
0345           MakeEntry and inserted into the local tree, and also links
0346           it into the supertree. }
0347           var
0348               place : PtrToEntry;
0349
0350               procedure FindLeaf;
0351               { FindLeaf searches the supertree to find where this
0352               node should be placed.  It will be appended to a leaf
0353               of course, and placed after entries with the same
0354               name. }
0355               var
0356                   subroot : PtrToEntry;
0357               begin
0358                   subroot := superroot;
0359                   while subroot <> nil do begin
0360                       place := subroot;
0361                       if savesymbol < subroot↑.procname then
0362                           subroot := subroot↑.before
0363                       else
0364                           subroot := subroot↑.after
0365                   end
0366               end; { FindLeaf }
0367
0368           begin { PutToSuperTree }
0369               if superroot = nil then begin
0370                   { Nothing in the supertree yet. }
0371                   superroot := newnode
0372               end else begin
0373                   { Seek the right place }
0374                   FindLeaf;
0375                   with place↑ do begin
0376                       if savesymbol < procname then
0377                           before := newnode
0378                       else
0379                           after := newnode
0380                   end
0381               end
0382           end; { PutToSuperTree }
0383
0384      begin { MakeEntry }
0385            located := false;
0386           savesymbol := symbol;
```

```
0387        if mainprog then begin
0388            new(newentry);
0389         end else if stack↑.scopetree = nil then begin
0390            { Nothing here yet. }
0391             new(newentry);
0392             stack↑.scopetree := newentry
0393         end else begin
0394            { Seek the identifier in the tree. }
0395             FindNode(located, node, stack↑.scopetree);
0396            if not located then begin
0397                { Normal case, make an entry. }
0398                 new(newentry);
0399                 with node↑ do
0400                     if symbol < procname then
0401                         left := newentry
0402                     else
0403                         right := newentry
0404             end
0405         end;
0406         if not located then begin
0407            { Here we initialize all the fields }
0408             with newentry↑ do begin
0409                 procname := symbol;
0410                caseset := symbolcase;
0411                 linenumber := lineno;
0412                startofbody := 0;
0413                if proc then
0414                    status := Shortform
0415                else
0416                    status := NotProc;
0417                 left   := nil;
0418                 right  := nil;
0419                before := nil;
0420                after  := nil;
0421                 calls := nil;
0422                localtree := nil
0423             end;
0424            MakeEntry := newentry;
0425            if proc then begin
0426                PutToSuperTree(newentry);
0427                 Push(newentry)
0428            end
0429         end else begin
0430            { Well, it'd better be forward or else. }
0431            MakeEntry := node;
0432             Push(node);
0433            if node↑.status = FwdHalf then begin
0434                stack↑.scopetree := node↑.localtree;
0435                node↑.status := AllFwd;
0436                node↑.forwardblock := lineno
0437            end else begin
0438                Error(5)
0439            end
0440        end
0441    end; { MakeEntry }
0442
0443    procedure PrintTree(root: PtrToEntry);
0444    var
0445         thiscell: ListOfUsages;
0446         count: Natural;
0447
0448        procedure ConditionalWrite(n: Natural;
0449                                   substitute: SixChars);
0450        begin
0451            { Write either the substitute string or a number. }
0452            if n = 0 then
0453                write(output, substitute)
0454            else
0455                write(output, n:6)
0456        end; { ConditionalWrite }
0457
0458        procedure NameWrite(p : PtrToEntry);
0459        var
0460            s : SetRange;
0461        begin
0462            for s := 0 to SetLimit do begin
0463                if s in p↑.caseset then
0464                    write(output,
0465                        chr(ord(p↑.procname[s+1])-UCLCdisplacement))
0466                else
0467                    write(output, p↑.procname[s+1])
0468             end
0469         end; { NameWrite }
0470
0471    begin { PrintTree }
0472        if root <> nil then
0473            with root↑ do begin
0474                PrintTree(before);
0475
0476                writeln(output);
0477                write(output, linenumber:5);
0478                ConditionalWrite(startofbody, '      ');
0479                case status of
0480                    FwdHalf,NotProc:
0481                        write(output, '   eh?');
0482                    Formal:
0483                        write(output, '   fml');
0484                    Outside:
0485                        write(output, '   ext');
0486                    Shortform:
0487                        write(output, '      ');
0488                    AllFwd:
0489                        write(output, forwardblock:6)
0490                end;
0491                write(output, '  ');
0492                NameWrite(root);
0493                write(output, ' :');
0494                thiscell := calls;
0495                count := 0;
0496                while thiscell <> nil do begin
0497                    if ((count mod namesperline) = 0) and (count <> 0)
0498                        then begin
0499                        writeln(output);
0500                        write(output, ' ':35, ' :')
0501                    end;
0502                    write(output, ' ');
0503                    NameWrite(thiscell↑.what);
0504                    thiscell := thiscell↑.next;
0505                    count := count + 1
0506                end;
0507                writeln(output);
0508
0509                PrintTree(after)
0510            end
0511    end; { PrintTree }
0512
0513    procedure NextToken;
0514    { This procedure produces the next "token" in a small set of
0515    recognized tokens.  Most of these serve an incidental purpose;
```

```
0516    the prime purpose is to recognize names (res'd words or identifiers).
0517    It serves also to skip dangerous characters in comments, strings,
0518    and numbers. }
0519
0520        procedure IgnoreComment;
0521        { This procedure skips over comments according to the definition
0522        in the Draft Pascal Standard. }
0523        begin
0524            NextCh;
0525            repeat
0526                while (ch <> '*') and (ch <> '}') do
0527                    NextCh;
0528                if ch = '*' then
0529                    NextCh;
0530            until (ch = ')') or (ch = '}');
0531            NextCh;
0532        end; { IgnoreComment }
0533
0534        procedure IgnoreNumbers;
0535        { This procedure skips numbers because the exponent part
0536        just might get recognized as a name!  Care must be taken
0537        not to consume half of a ".." occurring in a construct like
0538        "1..Name", or worse to consume it and treat the name as an
0539        possible exponent as in "1..E02". Ugh. }
0540        begin
0541            while ch in digits do
0542                NextCh;
0543            { The construction of NextCh, chno & line ensure that
0544            the following tests are always defined.  It is to get
0545            rid of tokens which begin with a period like .. & .)  }
0546            if (ch = '.') then begin
0547                if (line[chno+1] in digits) then begin
0548                    NextCh;
0549                    while ch in digits do
0550                        NextCh
0551                end;
0552            end;
0553            if (ch = 'E') or (ch = 'e') then begin
0554                NextCh;
0555                if (ch = '+') or (ch = '-') then
0556                    NextCh;
0557                while ch in digits do
0558                    NextCh
0559            end
0560        end; { IgnoreNumbers }
0561
0562        procedure ReadIdent;
0563        { This procedure reads in an identifier }
0564        var
0565            j : Positive;
0566        begin
0567            token := NameSy;
0568            symbol := Spaces;
0569            symbolcase := [];
0570            j := 1;
0571            while (j <= SigCharLimit) and (ch in alphanums) do begin
0572                if ch in uppercase then begin
0573                    symbol[j] := chr(ord(ch) + UCLCdisplacement);
0574                    symbolcase := symbolcase + [j-1]
0575                end else begin
0576                    symbol[j] := ch
0577                end;
0578                j := j+1;
0579                NextCh
0580            end;
0581            { In case there is a tail, skip it. }
0582            while ch in alphanums do
0583                NextCh
0584        end; { ReadIdent }
0585
0586    begin
0587        token := OtherSy;
0588        repeat
0589            if ch in usefulchars then begin
0590                case ch of
0591
0592                ')': begin
0593                        NextCh;
0594                        token := RParenSy
0595                    end;
0596
0597                '(': begin
0598                        NextCh;
0599                        if ch = '*' then begin
0600                            IgnoreComment
0601                        end else begin
0602                            token := LParenSy
0603                        end
0604                    end;
0605
0606                '{': begin
0607                        IgnoreComment
0608                    end;
0609
0610                '''': begin
0611                        NextCh;
0612                        while ch <> '''' do
0613                            NextCh;
0614                        NextCh
0615                    end;
0616
0617                '0','1','2','3','4','5','6','7','8','9':
0618                    begin
0619                        IgnoreNumbers
0620                    end;
0621
0622                ':': begin
0623                        NextCh;
0624                        if ch = '=' then begin
0625                            token := AssignSy;
0626                            NextCh
0627                        end else begin
0628                            token := ColonSy
0629                        end
0630                    end;
0631
0632                '.': begin
0633                        NextCh;
0634                        if ch <> '.' then
0635                            token := PeriodSy
0636                        else begin
0637                            token := SubRangeSy;
0638                            NextCh
0639                        end
0640                    end;
0641
0642                ';': begin
```

```
                        NextCh;
                        token := SemiColSy
                    end;

                'A','B','C','D','E','F','G','H','I','J','K','L','M',
                'N','O','P','Q','R','S','T','U','V','W','X','Y','Z',
                'a','b','c','d','e','f','g','h','i','j','k','l','m',
                'n','o','p','q','r','s','t','u','v','w','x','y','z':
                    begin
                        ReadIdent
                    end

                end
            end else begin
                { Uninteresting character }
                NextCh
            end
        until token <> OtherSy
    end; { NextToken }

    procedure ProcessUnit(programid: Boolean);
    { This procedure processes a program unit.  It is called on
    recognition of its leading token = program/procedure/function.
    The parameter records whether we currently have the main program
    identifier in the token, or not.  It doesn't have scope. }
    var
        at : PtrToEntry;

        function NameIsInScope: Boolean;
        { This function is called during the declaration phase
        of a block, and has to find any procedure which gets
        renamed by the scope rules. }
        var
            llevel      : PtrToStackCell;
            discovered  : Boolean;
            where       : PtrToEntry;
        begin
            llevel := stack;
            discovered := false;
            savesymbol := symbol;
            while (llevel <> nil) and not discovered do begin
                FindNode(discovered, where, llevel↑.scopetree);
                if not discovered then
                    llevel := llevel↑.substack
            end;
            if discovered then
                NameIsInScope := (where↑.status <> NotProc)
            else
                NameIsInScope := false
        end; { NameIsInScope }

        procedure ProcessBlock;
        { This procedure is called by ProcessUnit when it has recognized
        the start of a block.  It handles the processing of the block. }
        var
            address: PtrToEntry;

            procedure CrossReferencer;
            { CrossReferencer is called whenever we have a name which
            might be a call to a procedure or function.  The only way
            we tell is by looking in the table to see.  If it is, then
            the list of usages of the procedure we are in is scanned and
            possibly extended. }
            var
                newcell : ListOfUsages;
                ptr     : ListOfUsages;
                home    : PtrToEntry;
                slevel  : PtrToStackCell;
                found   : Boolean;

                procedure FindCell;
                { FindCell is used to scan a List Of Usages to determine
                whether the name already appears there.  If not, it
                leaves ptr pointing to the tail of the list so that an
                addition can be made. }
                var
                    nextptr  : ListOfUsages;
                begin
                    found := false;
                    nextptr := stack↑.current↑.calls;
                    if nextptr <> nil then
                        repeat
                            ptr := nextptr;
                            found := (ptr↑.what↑.procname = savesymbol);
                            nextptr := ptr↑.next
                        until found or (nextptr = nil)
                    else
                        ptr := nil
                end; { FindCell }

            begin { CrossReferencer }
                slevel := stack;
                found := false;
                while (slevel <> nil) and not found do begin
                    FindNode(found, home, slevel↑.scopetree);
                    if not found then
                        slevel := slevel↑.substack
                end;
                if found then begin
                    if home↑.status <> NotProc then begin
                        FindCell;
                        if not found then begin
                            new(newcell);
                            if ptr <> nil then
                                ptr↑.next := newcell
                            else
                                stack↑.current↑.calls := newcell;
                            newcell↑.what := home;
                            newcell↑.next := nil
                        end
                    end
                end
            end; { CrossReferencer }

            procedure ScanForName;
            { This procedure is required to go forward until the
            current token is a name (reserved word or identifier). }
            begin
                NextToken;
                while token <> NameSy do
                    NextToken
            end; { ScanForName }

        begin { ProcessBlock }
            while (symbol <> Sbegin) do begin
                while (symbol <> Sbegin) and (symbol <> Sprocedure) and
                    (symbol <> Sfunction) do begin
                    ScanForName;
```

```
0771                if NameIsInScope then begin
0772                    address := MakeEntry(false, false);
0773                    { MakeEntry made its status NotProc }
0774                end
0775            end;
0776            if symbol <> Sbegin then begin
0777                ProcessUnit(false);
0778                ScanForName
0779            end
0780        end;
0781        { We have now arrived at the body }
0782        depth := 1;
0783        stack↑.current↑.startofbody := lineno;
0784        NextToken;
0785        while depth <> 0 do begin
0786            if token <> NameSy then begin
0787                NextToken
0788            end else begin
0789                if (symbol = Sbegin) or (symbol = Scase) then begin
0790                    depth := depth + 1;
0791                    NextToken
0792                end else if (symbol = Send) then begin
0793                    depth := depth - 1;
0794                    NextToken
0795                end else begin
0796                    { This name is a candidate call.  But first we
0797                    must eliminate assignments to function values. }
0798                    savesymbol := symbol;
0799                    NextToken;
0800                    if token <> AssignSy then begin
0801                        CrossReferencer
0802                    end else begin
0803                        NextToken
0804                    end
0805                end
0806            end
0807        end
0808    end; { ProcessBlock }
0809
0810    procedure ScanParameters;
0811    { This procedure scans the parameter list because at the outer
0812    level there may be a formal procedure we ought to know about. }
0813    var
0814        which : PtrToEntry;
0815
0816        procedure ScanTillClose;
0817        { This procedure is called when a left parenthesis is
0818        detected, and its task is to find the matching right
0819        parenthesis.  It does this recursively. }
0820        begin
0821            NextToken;
0822            while token <> RParenSy do begin
0823                if token = LParenSy then
0824                    ScanTillClose;
0825                NextToken
0826            end
0827        end; { ScanTillClose }
0828
0829    begin { ScanParameters }
0830        NextToken;
0831        while token <> RParenSy do begin
0832            if (token = NameSy) then begin
0833                if (symbol = Sprocedure) or
0834                    (symbol = Sfunction) then begin
0835                    { A formal procedural/functional parameter. }
0836                    NextToken;
0837                    if token = NameSy then begin
0838                        which := MakeEntry(false, true);
0839                        which↑.status := Formal;
0840                        Pop;
0841                        NextToken;
0842                        if token = LParenSy then begin
0843                            { Skip interior lists. }
0844                            ScanTillClose
0845                        end
0846                    end else begin
0847                        Error(2);
0848                        NextToken
0849                    end
0850                end else begin
0851                    if NameIsInScope then
0852                        which := MakeEntry(false, false);
0853                    NextToken
0854                end
0855            end else begin
0856                NextToken
0857            end
0858        end;
0859        NextToken
0860    end; { ScanParameters }
0861
0862 begin { ProcessUnit }
0863    printflag := true;
0864    adjustment := First;
0865    NextToken;
0866    if token <> NameSy then
0867        Error(2)
0868    else begin
0869        { We now have the name to store away. }
0870        at := MakeEntry(programid, true);
0871        while not (token in [LParenSy,SemiColSy,ColonSy]) do
0872            NextToken;
0873        if token = LParenSy then
0874            ScanParameters;
0875        while token <> SemiColSy do
0876            NextToken;
0877        PrintLine;
0878        { We have now printed the procedure heading. }
0879        printflag := false;
0880        writeln(output);
0881        { Our next task is to see if there is an attached block. }
0882        NextToken;
0883        if token <> NameSy then
0884            Error(3)
0885        else begin
0886            if (symbol <> Slabel) and (symbol <> Sconst) and
0887                (symbol <> Stype) and (symbol <> Sprocedure) and
0888                (symbol <> Sfunction) and (symbol <> Svar) and
0889                (symbol <> Sbegin) then begin
0890                { Bloody directive, mate. }
0891                if symbol = Sforward then
0892                    at↑.status := FwdHalf
0893                else
0894                    at↑.status := Outside;
0895                Pop
0896            end else begin
0897                ProcessBlock;
0898                Pop
```

```
0899                end
0900              end
0901            end
0902      end; { ProcessUnit }
0903
0904 { *** ------------------------------------------------------¦
0905   ¦                                                         ¦
0906   ¦   This procedure outlines what is needed  to insert the ¦
0907   ¦   predefined names into Referencer's tables.  De-box it ¦
0908   ¦   and extend it as needed.                              ¦
0909   ¦                                                         ¦
0910   ¦   procedure BuildPreDefined;                           ¦
0911   ¦   const                                                 ¦
0912   ¦       NoOfNames = 2;                                    ¦
0913   ¦   type                                                  ¦
0914   ¦       NamesIndex   = 1..NoOfNames;                      ¦
0915   ¦   var                                                   ¦
0916   ¦       kk   : NamesIndex;                                ¦
0917   ¦       tt   : array[NamesIndex] of PseudoString;         ¦
0918   ¦       hohum: PtrToEntry;                                ¦
0919   ¦   begin                                                 ¦
0920   ¦       tt[01] := 'new             ';                     ¦
0921   ¦       tt[02] := 'writeln         ';                     ¦
0922   ¦       caseset := [];                                    ¦
0923   ¦       for kk := 1 to NoOfNames do begin                 ¦
0924   ¦           symbol := tt[kk];                             ¦
0925   ¦           hohum := MakeEntry(false,false);              ¦
0926   ¦           hohum↑.status := Outside;                     ¦
0927   ¦       end;                                              ¦
0928   ¦   end;                                                 ¦
0929   ¦                                                         ¦
0930   ¦------------------------------------------------------ *** }
0931
0932      procedure PrintHeading;
0933      begin
0934           writeln(output, 'Procedural Cross-Referencer - Version S-02.01');
0935           writeln(output, '==========================================');
0936           writeln(output)
0937      end; { PrintHeading }
0938
0939 begin { Referencer }
0940      superroot := nil;
0941      { Here we construct an outer-scope stack entry.  This is needed
0942      to hold any pre-defined names.  The distributed version does not
0943      include any of these, but they are easily provided.  See the
0944      outlines in the code marked with *** if you want this feature. }
0945      new(stack);
0946      with stack↑ do begin
0947           current := nil;
0948           scopetree := nil;
0949           substack := nil
0950      end;
0951
0952      printflag := false;
0953
0954      uppercase := ['A','B','C','D','E','F','G','H','I','J','K','L','M',
0955                    'N','O','P','Q','R','S','T','U','V','W','X','Y','Z'];
0956      alphabet  := uppercase +
0957                   ['a','b','c','d','e','f','g','h','i','j','k','l','m',
0958                    'n','o','p','q','r','s','t','u','v','w','x','y','z'];
0959      digits    := ['0','1','2','3','4','5','6','7','8','9'];
0960      alphanums := alphabet + digits { *** + ['_'] *** } ;
0961      usefulchars := alphabet + digits +
0962                     ['(', ')', '{', '.', ':', ';', ''''];
```

```
0963
0964      namesperline := (LineWidth - (SigCharLimit + 21)) div
0965           (SigCharLimit + 1);
0966
0967      { *** If you want to introduce some options, this is the place
0968      to insert the call to your OptionAnalyser.  None is provided
0969      with the standard tool because the requirements vary widely
0970      across user environments.  The probable options that might be
0971      provided are (a) whether pre-declared names should appear in
0972      the call lists, (b) how many columns are to be printed in them
0973      (namesperline), (c) whether underscore is permitted in identifiers,
0974      and perhaps whether output should be completely in upper-case
0975      letters.  The first option (a) requires a call to BuildPreDefined
0976      just below this point, after analysing options... }
0977
0978      total := 0;
0979      chno := 0;
0980      lineno := 0;
0981      level := -1;
0982      errorflag := false;
0983      { *** BuildPreDefined; *** }
0984
0985      { *** page(output); *** }
0986      PrintHeading;
0987      writeln(output, ' Line   Program/procedure/function heading');
0988      for pretty := 1 to 43 do
0989           write(output, '-');
0990      writeln(output);
0991      writeln(output);
0992      { Now we need to get the first token, which should be program. }
0993      NextToken;
0994      if token <> NameSy then
0995           Error(1)
0996      else if symbol <> Sprogram then
0997           Error(1)
0998      else begin
0999           ProcessUnit(true);
1000           { Having returned, there ought to be a period here. }
1001           if not errorflag then begin
1002                { We check all tokens that begin with a period because
1003                what occurs after the closing period is nothing to do
1004                with us. }
1005                if (token <> PeriodSy) and (token <> SubRangeSy) then
1006                     Error(4)
1007                else begin
1008                     adjustment := First;
1009                     PrintLine
1010                end
1011           end
1012      end;
1013      { Completed Phase One - now for the next. }
1014      if not errorflag then begin
1015           page(output);
1016           PrintHeading;
1017           writeln(output,
1018                ' Head  Body  Notes ',
1019                ' ':SigCharLimit,
1020                '   Calls made to');
1021           for pretty := 1 to (SigCharLimit+37) do
1022                write(output, '-');
1023           writeln(output);
1024           PrintTree(superroot);
1025           writeln(output)
1026      end
1027 end.
```

AN OVERVIEW OF MAP

MAP provides four basic additions to Pascal: constant expression evaluation; source file inclusion; parameterized macro substitution; and conditional compilation. This section contains a discussion of each of these facilities.

MAP evaluates constant expressions (expressions where operands are constants or previously defined symbolic constants) on the right-hand side of CONST declarations. Expressions may contain the following operators (listed in descending precedence):

```
function:       name (arguments)
negating:       NOT -
multiplying:    AND * / DIV MOD MIN MAX
adding:         OR + -
relating:       < <= = <> >= >
concatenating:  (one or more blanks)
```

All standard operators have the same meaning as in Pascal, and strong typing is observed. The operators MIN and MAX require operands of type INTEGER or REAL and return the smaller and larger of their operands, respectively. Concatenation requires operands of type PACKED ARRAY OF CHAR, and returns a PACKED ARRAY OF CHAR which is their concatenation (the type CHAR is assumed to be a packed array of one character for concatenation).

MAP recognizes the standard Pascal functions ABS, SQR, CHR, ORD, ROUND, TRUNC, as well as two nonstandard functions, LENGTH and STRINGOF. LENGTH requires an argument of type PACKED ARRAY OF CHAR or CHAR, and returns the number of characters in it. STRINGOF requires an integer argument, and returns a PACKED ARRAY OF CHAR consisting of its decimal representation.

Operands in CONST expressions may be constants or previously defined CONST names. Of course, Pascal scope rules apply to defined names. MAP also provides several predefined symbolic constants which can be used in CONST expressions. Two especially useful predefined names, TIME and DATE, give the time and date on which the compilation was performed. These predefined constants help when writing production programs that must be time and date stamped. For example, in a production program a heading is usually printed whenever the program runs:

```
'PROGRAM XYZ COMPILED ON mm/dd/yy AT hh:mm:ss'
```

Such a heading may provide the only link between an object version of a program and its source. Unfortunately, a programmer may fail to update the heading when making changes to the program. Using the predefined constants in MAP to create the heading relieves the programmer of the updating task and guarantees the heading will always be accurate:

```
CONST
  READING = 'PROGRAM XYZ COMPILED ON' DATE 'AT' TIME;
```

In addition to constant expression evaluation, MAP supplies a macro substitution facility. A macro, which may have zero or more formal parameters, may be defined anywhere in the source program using the syntax:

```
$DEFINE(name(formals),value)
```

where 'name' is a valid Pascal identifier, 'formals' is a list of identifiers separated by commas, and 'value' is a sequence of Pascal tokens which is well balanced with respect to parentheses. Once a macro has been defined, it can be called by coding

```
$name(actuals)
```

where 'name' is the name of the macro, and 'actuals' is a list of actual parameters separated by commas. Each actual parameter must be a sequence of Pascal tokens which is well balanced with respect to parentheses.

In addition to the user-defined macros, MAP recognizes several system macros. Definition of a new macro, as shown above, requires the use of the one such system macro, DEFINE. Another system macro, INCLUDE, provides for source file inclusion. When MAP encounters a call:

```
$INCLUDE(file name)
```

it opens the named file, and continues processing, reading input from the new file. Upon encountering an end-of-file condition, MAP closes the included file, and resumes processing the original file. Includes may be nested, but they may not be recursive (even though there is a way to prevent an infinite recursion).

One may think of 'include' as a macro whose body is an entire file. This view, however, does not reflect the fact that the user also expects included text to be listed like standard input rather than like the body of a macro. While macro expansions are not usually displayed in the source listing, included files are. Therefore, INCLUDE has a special status among macros.

One other system macro, CODEIF, is provided to support the conditional compilation of code. The syntax of CODEIF is:

```
$CODEIF(constant Boolean expression,code)
```

where the constant Boolean expression follows the rules for CONST expressions outlined above, and code represents a sequence of Pascal tokens which is well balanced with respect to parentheses. If the Boolean expression evaluates to 'true', the code is compiled; if the expression evaluates to 'false', the code is skipped.

REFERENCE

1. D. Comer, 'A Pascal Macro Preprocessor for Large Program Development', Software Practice and Experience, vol. 9, 203-209 (1979).

```
program map(output, psource);

{ ==============  portable version  ============================== }
{ ********************************************************************** }
{                                                                        }
{   program :       M A P (Macro Pascal) -- Pascal preprocessor with     }
{                   constant expressions, macros, included files, and    }
{                   conditional compilation.  (portable version)         }
{                                                                        }
{   date :          February 12, 1978, modified April 30, 1979           }
{                                                                        }
{   programmer :    Doug Comer, Computer Science Department, Purdue      }
{                                                                        }
{   input :         A Pascal program with expressions allowed in the     }
{                   const values, and macro definitions and calls.       }
{                   Macros may be called from the source code by         }
{                   writing the name prefixed with a dollar sign, with   }
{                   actual parameters supplied as a string               }
{                   enclosed in parentheses.  The actual parameters      }
{                   may not contain references to other actual           }
{                   parameters or macros.  Formal parameter references,  }
{                   also denoted by $name in the body of the macro,      }
{                   override macro definitions, so a macro with formal   }
{                   'a' cannot call macro 'a'.  Null argument lists      }
{                   like () must be used when calling a macro with no    }
{                   actual parameters.  Null parameters will be used     }
{                   if insufficient actual parameters are specified;     }
{                   extra actuals are ignored.  Note that this differs   }
{                   from the version cited in the paper.                 }
{                   Input must be in columns 1 - 'rc' (default 72).       }
{                                                                        }
{   output :        Output is the file, psource, a compressed version    }
{                   of the Pascal source deck.  The present version      }
{                   strips all comments except '(*$' and all the         }
{                   unnecesary blanks in performing the compression.     }
{                   Also, the source is crammed into 'prc' columns,      }
{                   the default being 71.                                }
{                                                                        }
{   system :        Pascal on CDC 6500, Purdue dual MACE                 }
{                                                                        }
{   Copyright :     (C) 1978.  Permission to copy, modify and            }
{                   distribute, but not for profit, is hereby granted,   }
{                   provided that this note is included.                 }
{                                                                        }
{ ********************************************************************** }

label 1   {  for aborting  };

const
   arrow     = '^';          {  pointer for errors  }
   blank     = ' ';
   break     = '     ';      {  break between rc and rest of line  }
   comma     = ',';
   defexpr   = true;         {  default is expression evaluation  }
   deflist   = true;         {  default is listing  }
   defprc    =  71;          {  default right column for pascal  }
   defrc     =  72;          {  default right column for map input  }
   dollar    = '$';
   double    = '0';          {  double space carriage control  }
   equal     = '=';
   errflag   = '                    ';
   errprefix = '---> error  ';
   errlen    =  40;          {  length of error message  }

   {    error messages    }

   erabstype = 'evalabs    - type error, number needed  ';
   erarith   = 'arith      - bad type                   ';
   eratntype = 'evalatn    - type error, number needed  ';
   erbodyeof = 'getbody    - end of file in macro body  ';
   erchrtype = 'evalchr    - type error, integer needed ';
   ercklpar  = 'ckmacro    - left paren expected        ';
   erckrpar  = 'ckmacro    - right paren expected       ';
   ercodcom  = 'docodeif   - syntax error, missing comma';
   ercodeof  = 'docodeif   - unexpected end of file     ';
   ercodtype = 'docodeif   - type error, boolean needed ';
   erconvert = 'convert    - integer truncated          ';
   ercostype = 'evalcos    - type error, number needed  ';
   erdefcom  = 'dodefine   - missing comma              ';
   erdefname = 'dodefine   - syntax error, name needed  ';
   erexptype = 'expression - invalid operand type       ';
   erextype  = 'evalexp    - type error, number needed  ';
   erfacrpar = 'factor     - right paren expected       ';
   erfactype = 'factor     - type conflict              ';
   erincname = 'doinclude  - file name needed           ';
   erincrpar = 'doinclude  - right paren expected       ';
   erindrpar = 'doindex    - right paren expected       ';
   erindxtyp = 'doindex    - type error, integer needed ';
   erlentype = 'evallen    - type error, string needed  ';
   erlntype  = 'evalln     - type error, number needed  ';
   erlongstr = 'gettok     - string exceeds source line ';
   ermacname = 'gettok     - illegal macro name         ';
   ermacdefn = 'getbsu     - undefined macro call       ';
   ermconsyn = 'parsemcon  - semicolon expected         ';
   eroctdig  = 'gettok     - illegal octal digit        ';
   eroddtype = 'evalodd    - type error, integer needed ';
   eropen    = 'open       - recursive includes ignored ';
   eropttype = 'dooptions  - error in options list      ';
   erordarg  = 'evalord    - ord requires 1 char. arg.  ';
   erordtype = 'evalord    - type error, char. needed   ';
   erover    = 'over       - table overflow             ';
   erparscon = 'parsecon   - equal sign needed          ';
   erparsend = 'parse      - unmatched end              ';
   erparseof = 'parse      - unexpected end of file     ';
   erparsfwd = 'parse      - unmatched forward decl.    ';
   erparsmcon= 'parsemcon  - equal sign needed          ';
   erpconsyn = 'parsecon   - semicolon expected         ';
   erputtok  = 'puttok     - token too large            ';
   errelatyp = 'relate     - illegal type for rel. oper.';
   errelconf = 'relate     - type conflict in relation  ';
   erroutype = 'evalrou    - type error, real needed    ';
   ersintype = 'evalsin    - type error, number needed  ';
   ersqrtype = 'evalsqr    - type error, number needed  ';
   erstrtype = 'evalstr    - type error, integer needed ';
   ersyslpar = 'dosysmac   - left paren expected        ';
   ertermtyp = 'term       - invalid operand type       ';
   ertrutype = 'evaltru    - type error, real needed    ';
   ervalexp  = 'variable   - value or name expected     ';
   ervarfnct = 'variable   - unknown function, 0 used   ';
   ervarrpar = 'variable   - right paren expected       ';

   greater   = '>';
   inname    = 'INPUT     ';   {  standard input file name  }
   inlname   = '          ';   {  standard input file name for   }
                                {  listing  }
   letterb   = 'B';
   lettere   = 'E';
   lparen    = '(';
   maxcalls  =  15;            {  max macro call depth  }
   maxcons   = 200;            {  max active const defns  }
   maxcol    = 120;            {  max right column for input/output  }
   maxcstr   =1000;            {  max const string area  }
   maxdefs   = 100;            {  max defined macros  }
   maxdefstr =4000;            {  max macro string area  }
   maxfiles  =   5;            {  max included file depth  }
   maxfns    =  14;            {  max recognized functions  }
   maxkeys   =  21;            {  max recognized language keywords  }
   maxline   = 140;            {  max characters per input line  }
   mincol    = 70;             {  min right column for input/output  }
   minus     = '-';
   ndefconst =   9;            {  number of predefined constants  }
   {} newline   = chr(10);        {  set to newline character  }
   newpage   = '1';            {  newpage carriage control  }
   nsysmac   =   5;            {  number of system macros  }
   pagesize  =  55;            {  lines/page not counting heading  }
   period    = '.';
   plus      = '+';
   quote     ='''';
   rparen    = ')';
   semi      = ';';
   space     = ' ';            {  single space carriage control  }
   star      = '*';
   sysinc    =   1;            {  codes for system macros  }
   syscodeif =   2;
   sysindex  =   3;
   sysdefine =   4;
   sysoption =   5;
   title1    = ' M A P    (vers 2.0p of 4/30/29)            ';
    title1a   = '                    run on ';
   title1b   = ' at ';
   title2    = '          include     pascal';
   title3    = ' line  file     line   line            source';
   title4    = ' ----  -------------  ------  ----------------';
   title5    = '------------------------------------------------';
   title6    = '-----';
   zero      = '0';


type

   alfa      = packed array[1..10] of char;
   text      = file of char;

   crng      = 0..maxcons;     {  constant expression stack   }
   csrng     = 0..maxcstr;     {  constant expr. string area  }
   drng      = 0..maxdefs;     {  macro definition stack      }
   dsrng     = 0..maxdefstr;   {  macro def. string area      }
   flrng     = 0..maxfiles;    {  included file stack         }
   fnrng     = 0..maxfns;      {  builtin functions           }
   krng      = 0..maxkeys;     {  keywords                    }
   lnrng     = 0..maxline;     {  input line                  }
   mrng      = 0..maxcalls;    {  macro call stack            }
   pgrng     = 0..pagesize;    {  listing page                }

   msg       = packed array[1..40] of char;

   fptr      =^formal;

   formal    = record
                  fname : alfa;  {  name of formal parameter  }
                  fnext : fptr
                  end;


   fns       = (fabs,fatn,fchr,fcos,fexp,   {  builtin functions  }
                flen,fln,fodd,ford,frou,fsin,fsqr,fstr,ftru);

   lex       = (lexadd,lexsub,   {  order dependent  }
                lexand,lexmult,lexdvd,lexmin,lexmax,lexdiv,lexmod,
                lexalpha,lexint,lexreal,lexst,lexmac,
                lexbeg,lexcas,lexend,lexrec,lexfun,lexproc,lexcon,
                lexmcon,
                lextpe,lexvar,lexfwd,
                lexor,lexnot,
                lexlt,lexle,lexeq,lexgt,lexge,lexne,
                lexsemi,lexother,
                lexlparen,lexrparen,
                lexcomma,lexeof);

   aptr      =^arg;

   arg       = record       {  actual argument list node  }
                  aform :  alfa; {  formal name             }
                  afirst : dsrng;{  start of actual in dstr  }
                  alast :  dsrng;
                  anext :  aptr
                  end;


   constyp   = (tbl,tch,terr,tin,tot,tre); { type of const expression }
```

```
221
222     cset      = set of constyp;
223
224     strng     = array[lnrng] of char;
225
226     errmsg    = packed array[1..errlen] of char;
227
228 var
229     ctab      : array[crng] of   { constant table }
230                   record
231                     cname : alfa;
232                     case ctyp : constyp of
233                       tin : (ci : integer);
234                       tre : (cr : real);
235                       tch : (cfirst : csrng; clen : csrng);
236                       tbl : (cb : boolean);
237                       tot : (co : alfa)
238                     end;
239     ctop,           { current top of ctab and last const }
240     cvalid    : crng;  { last nontemporary constant }
241
242     cstr      : array[csrng] of char;{ string const storage }
243     cstop     : csrng;
244
245     fstack    : array[flrng] of      { included file stack }
246                   record
247                     fname : alfa;   { file name              }
248                     ffile : text;
249                     fline : integer
250                     end;
251     ftop      : -1..maxfiles;
252
253     keywd     : array[0..maxkeys] of { language keywords }
254                   record
255                     kname : alfa;  { keyword name }
256                     klex :  lex
257                     end;
258
259     mstack    : array[mrng] of  { macro calls }
260                   record
261                     margs :  aptr;   { list of arguments       }
262                     mnext :  dsrng;  { next char to read       }
263                     mlast :  dsrng;  { last char in this macro }
264                     matop :  dsrng   { actual top upon call    }
265                     end;
266
267     mtop      : mrng;       { top of called macro stack }
268
269     defs      : array[drng] of    { macro definitions }
270                   record
271                     dname : alfa;   { macro name              }
272                     dfirst: dsrng;  { first char in this macro }
273                     dlast : dsrng;  { last char in this macro  }
274                     dargs : fptr    { list of formals         }
275                     end;
276
277     dtop      : drng;
278
279     defstr    : array[dsrng] of char;  { macro definition bodies }
280
281     dstop     : dsrng;      { top of definition string area }
282
283     atop      : dsrng; { actual arguments saved in top of defstr }
284
285     funct     : array[fnrng] of   { list of builtin functions }
286                   record
287                     fnnme : alfa;  { function name }
288                     fntyp : fns
289                     end;
290
291     inline    : strng;    { input line }
292     last,
293     next      : lnrng;    { last char and next char in inline }
294     ch        : char;     { next character from getch }
295     line      : integer;  { last line number }
296     pline     : integer;  { next pascal output line number }
297
298     tme,                  { time of day from system }
299     dte       : alfa;     { date from system }
300     timein    : integer;  { clock value at start of run }
301     tottme    : integer;  { total time used in ms }
302
303     linectr   : integer;  { lines so far on this page }
304     nerrors   : integer;  { number of errors found }
305
306     psource,
307     dummy     : text;   { dummy used for real number conversion }
308
309     rcopt,
310     prcopt    : lnrng;    { right column on input/output }
311     listopt   : boolean;  { list on or off }
312     expropt   : boolean;  { recognize expressions on or off }
313
314     lastlex   : lex;      { last token type put by puttok }
315     outpos    : lnrng;    { last column pos used by puttok }
316
317      lexstr    : strng;    { lexical string }
318     lexlen    : lnrng;    { number of chars in lexstr }
319     lextyp    : lex;      { type of token in lexstr }
320
321     index     : integer;  { for $index macro }
322
323     confl     : set of lex;
324                           { set of tokens needing blank between }
325
326 {  forward declarations for all procedures and functions }
327
328 procedure arith;   forward;
329  procedure ckformal(name: alfa; { formal name } var found: boolean);
330   forward;
331  procedure ckmacro(name: alfa; { macro name } var found: boolean);
332   forward;
333  procedure close;   forward;
334  procedure convrt;   forward;
335  procedure convrti;   forward;
336  procedure convrtr;   forward;
337  procedure convrts;   forward;
338  procedure docodeif;   forward;
339  procedure dodefine;   forward;
340  procedure doinclude;   forward;
341  procedure doindex;   forward;
342  procedure dooptions;   forward;
343  procedure dosysmac(d: drng);  { which macro }    forward;
344  procedure error(err: errmsg);   forward;
345  procedure evalfns(f: fns);   forward;
346  procedure evalabs;   forward;
347  procedure evalatn;   forward;
348  procedure evalchr;   forward;
349  procedure evalcos;   forward;
350  procedure evalexp;   forward;
351  procedure evallen;   forward;
352  procedure evalln;   forward;
353  procedure evalodd;   forward;
354  procedure evalord;   forward;
355  procedure evalrou;   forward;
356  procedure evalsin;   forward;
357  procedure evalsqr;   forward;
358  procedure evalstr;   forward;
359  procedure evaltru;   forward;
360  procedure experror(err: errmsg);   forward;
361  procedure expression;   forward;
362  procedure factor;   forward;
363  procedure findcon(name: alfa;
364 {  name of const } var found: boolean);   forward;
365  procedure flookup(name: alfa;
366 {  function name } var fun: fns;
367 {  function code } var found: boolean);   forward;
368  procedure flush;   forward;
369  procedure forcereal;   forward;
370  procedure getactuals(f: fptr;
371 {  pointer to next formal } var act: aptr); {  pointer to actual }
372    forward;
373  procedure getbody;   forward;
374  procedure getbsu;   forward;
375  procedure getcdparm;   forward;
376  procedure getch;   forward;
377  procedure getformals(var f: fptr);   forward;
378  procedure getkey;   forward;
379  procedure getline;   forward;
380  procedure getparm;   forward;
381  procedure gettok;   forward;
382  procedure initialize;   forward;
383  procedure need(l: pgrng);   forward;
384  procedure newpg;   forward;
385  procedure open(name: alfa);  { file name to open }   forward;
386  procedure over(i: integer;
387 {  current value } maxval: integer);  {  max value }   forward;
388  procedure parse(top: crng;
389 {  ctop upon entry } tok: lex); {  token causing recursion }
390   forward;
391  procedure parsecon;   forward;
392  procedure parsemcon;   forward;
393  procedure pushback;   forward;
394  procedure puttok;   forward;
395  procedure relate;   forward;
396  procedure scanheader;   forward;
397  procedure term;   forward;
398  procedure terminate;   forward;
399  procedure timedate;   forward;
400  function typesmatch: boolean;   forward;
401  function typeis(c: cset): boolean;   forward;
402  procedure variable;   forward;
403
404 {  procedures and functions }
405
406
407 { ******** }
408 {  arith - recognize arithmetic ops in expression }
409 { ******** }
410  procedure arith;
411
412   var
413                op: lex;
414
415   begin
416     term;
417     if (lextyp in [lexor, lexadd, lexsub]) and (not typeis([terr]))
418     then
419       if ((lextyp = lexor) and typeis([tbl])) or ((lextyp in [lexadd,
420          lexsub]) and typeis([tin, tre]))
421       then
422         begin
423           over(ctop, maxcons);
424           while lextyp in [lexor, lexadd, lexsub] do
425             begin
426               ctop := ctop + 1;   op := lextyp;   getkey;   term;
427               if (op = lexor) and typeis([tbl])
428               then with ctab[ctop - 1] do cb := cb or ctab[ctop].cb
429               else
430                 if (op in [lexadd, lexsub]) and typeis([tin, tre])
431                 then
432                   with ctab[ctop - 1] do
433                     if (ctyp = tin) and (ctab[ctop].ctyp = tin)
434                     then
435                       case op of
436                         lexadd: ci := ci + ctab[ctop].ci;
437                         lexsub: ci := ci - ctab[ctop].ci
438                       end {  case }
439                     else
440                       begin
```

```
441                     forcereal;
442                     case op of
443                       lexadd: cr := cr + ctab[ctop].cr;
444                       lexsub: cr := cr - ctab[ctop].cr
445                     end {  case  }
446                   end
447               else
448                 if ctab[ctop].ctyp <> terr   then experror(erarith);
449             ctop := ctop - 1
450           end
451       end
452   end {  arith  };
453
454  { ******** }
455  {  ckformal - if reference to formal, push on call stack  }
456  { ******** }
457   procedure ckformal { name:alfa; var found:boolean };
458
459    var
460                   a: aptr;
461
462    begin
463      found := false;
464      if mtop > 0
465      then
466        begin
467          a := mstack[mtop].margs;
468          while (a <> nil) and (not found) do
469            begin
470              with a^ do
471                if aform = name
472                then
473                  begin
474                    found := true;   pushback;   mtop := mtop + 1;
475                    with mstack[mtop] do
476                      begin
477                        margs := nil;   mnext := afirst;   mlast := alast;
478                        matop := atop
479                      end;
480                    getch
481                  end;
482              a := a^.anext
483            end;
484          if found   then gettok
485        end
486    end {  ckformal  };
487
488  { ******** }
489  {  ckmacro - if macro called, push onto stack  }
490  { ******** }
491   procedure ckmacro { name:alfa; var found:boolean };
492
493    var
494                   d: drng {  index to defined macros  };
495
496    begin
497      d := dtop;   defs[0].dname := name;
498      while defs[d].dname <> name do d := d - 1;
499      if d > 0
500      then
501        begin
502          found := true;
503          if d <= nsysmac   then dosysmac(d)
504          else
505            begin
506              over(mtop, maxcalls);
507              with mstack[mtop + 1], defs[d] do
508                begin
509                  margs := nil;   mnext := dfirst;   mlast := dlast;
510                  matop := atop;   while ch = blank do getch;
511                  if ch = lparen
512                  then
513                    begin
514                      getch;   getactuals(dargs, margs);
515                      if ch <> rparen   then error(erckrpar)
516                    end
517                  else error(ercklpar)
518                end;
519              mtop := mtop + 1;   getch
520            end;
521          gettok
522        end
523    end {  ckmacro  };
524
525  { ******** }
526  {  close - close the current file + restore old one  }
527  { ******** }
528   procedure close;
529
530    begin ftop := ftop - 1 end {  close  };
531
532  { ******** }
533  {  convrt - convert constant to pascal input format  }
534  { ******** }
535   procedure convrt;
536
537    var
538                   i: integer;
539                   c: char;
540                sign: boolean;
541
542    begin
543      with ctab[ctop] do
544        case ctyp of
545          tin:
546            begin
547              if abs(ci) >= maxint
548              then begin i := maxint;   error(erconvert) end
549              else i := ci;
550              if i < 0   then begin sign := true;   i := abs(i) end
551              else sign := false;
552              lexlen := 0;
553              while i > 0 do
554                begin
555                  lexlen := lexlen + 1;
556                  lexstr[lexlen] := chr(ord('0') + (i mod 10));
557                  i := i div 10
558                end;
559              if sign then
560                begin lexlen := lexlen + 1;   lexstr[lexlen] := minus end;
561              for i := 1 to(lexlen div 2) do
562                begin
563                  c := lexstr[i];   lexstr[i] := lexstr[lexlen - i + 1];
564                  lexstr[lexlen - i + 1] := c
565                end;
566              lextyp := lexint
567            end;
568          terr:;
569          tot:
570            begin
571              lexlen := 10;   unpack(co, lexstr, 1);   lextyp := lexalpha;
572              while lexstr[lexlen] = blank do lexlen := lexlen - 1
573            end;
574          tch:
575            begin
576              lextyp := lexst;   lexlen := 1;   lexstr[1] := quote;
577              for i := 0 to clen - 1 do
578                begin
579                  lexlen := lexlen + 1;
580                  lexstr[lexlen] := cstr[cfirst + i];
581                  if lexstr[lexlen] = quote then
582                    begin lexlen := lexlen + 1;   lexstr[lexlen] := quote
583                    end
584                end;
585              lexlen := lexlen + 1;   lexstr[lexlen] := quote
586            end;
587          tbl:
588            begin
589              lextyp := lexalpha;
590              if cb
591              then begin unpack('TRUE', lexstr, 1);   lexlen := 4 end
592              else begin unpack('FALSE', lexstr, 1);   lexlen := 5 end
593            end;
594          tre:
595            begin
596              rewrite(dummy);   write(dummy, cr, blank);   reset(dummy);
597              while dummy^ = blank do get(dummy);   lexlen := 0;
598              while dummy^ <> blank do
599                begin
600                  lexlen := lexlen + 1;   lexstr[lexlen] := dummy^;
601                  get(dummy)
602                end;
603              lextyp := lexreal
604            end
605        end {  case  }
606    end {  convrt  };
607
608  { ******** }
609  {  convrti - convert integer token to binary form  }
610  { ******** }
611   procedure convrti;
612
613    var
614                   i: integer;
615                   l: lnrng;
616
617    begin
618      with ctab[ctop] do
619        begin
620          ctyp := tin;   ci := 0;
621          for l := 1 to lexlen do
622            ci := 10 * ci + ord(lexstr[l]) - ord(zero)
623        end
624    end {  convrti  };
625
626  { ******** }
627  {  convrtr - convert real token to binary form  }
628  { ******** }
629   procedure convrtr;
630
631    var
632                   i: lnrng;
633
634    begin
635      rewrite(dummy);   for i := 1 to lexlen do write(dummy, lexstr[i]);
636      write(dummy, blank);   reset(dummy);
637      with ctab[ctop] do begin ctyp := tre;   read(dummy, cr) end
638    end {  convrtr  };
639
640  { ******** }
641  {  convrts - convert quoted string to const string  }
642  { ******** }
643   procedure convrts;
644
645    var
646                   l: lnrng;
647
648    begin
649      with ctab[ctop] do
650        begin
651          ctyp := tch;   clen := 0;   cfirst := cstop + 1;
652          l := 2 {  skip leading quote  };
653          while l <= (lexlen - 1) do
654            begin
655              clen := clen + 1;   over(cstop, maxcstr);
656              cstop := cstop + 1;   cstr[cstop] := lexstr[l];
657              if lexstr[l] = quote   then l := l + 2   else l := l + 1
658            end
659        end
660    end {  convrts  };
```

```
661
662  { ******** }
663  {  docodeif - process $codeif( expr., code)    }
664  { ******** }
665   procedure docodeif;
666
667    var
668                    a: dsrng {  save area for atop upon entry  };
669                  ctr: integer {  left paren count  };
670
671    begin
672      getkey;   over(ctop, maxcons);   ctop := ctop + 1;   expression;
673      ctop := ctop - 1;   a := atop;
674      if lextyp <> lexcomma   then experror(ercodcom)
675      else
676        with ctab[ctop + 1] do
677          if ctyp = tbl
678          then
679            if cb
680            then
681              begin
682                over(mtop, maxcalls);
683                with mstack[mtop + 1] do
684                  begin
685                    margs := nil;   mlast := atop - 1;   getcdparm;
686                    mnext := atop;   matop := a;
687                  end;
688                mtop := mtop + 1;   getch
689              end
690            else
691              begin
692                ctr := 1;
693                while ctr > 0 do
694                  begin
695                    if ch = newline
696                    then
697                      begin
698                        if (mtop = 0) and (ftop = 0) and eof(fstack[0].
699                                                             ffile)
700                          then begin error(ercodeof);   goto 1 end
701                      end
702                    else
703                      if ch = rparen   then ctr := ctr - 1
704                      else if ch = lparen   then ctr := ctr + 1;
705                    getch
706                  end
707              end
708        else if ctyp <> terr   then error(ercodtype)
709    end { docodeif };
710
711  { ******** }
712  {  dodefine - process $define(name(formal parms),string)   }
713  { ******** }
714   procedure dodefine;
715
716    begin
717      gettok;
718      if lextyp <> lexalpha   then error(erdefname)
719      else
720        begin
721          over(dtop, maxdefs);   dtop := dtop + 1;
722          with defs[dtop] do
723            begin
724              lexstr[0] := dollar;   pack(lexstr, 0, dname);
725              dfirst := dstop + 1;   dlast := dstop;   gettok;
726              if lextyp = lexlparen
727              then begin gettok;   getformals(dargs);   gettok end
728              else dargs := nil
729            end;
730          if lextyp <> lexcomma
731          then begin error(erdefcom);   dtop := dtop - 1 end
732          else getbody
733        end
734    end {  dodefine  };
735
736  { ******** }
737  {  doinclude - process $include(file)   }
738  { ******** }
739   procedure doinclude;
740
741    var
742                name: alfa;
743
744    begin
745      getbsu;
746      if lextyp <> lexalpha   then error(erincname)
747      else
748        begin
749          pack(lexstr, 1, name) {  check file name here if desired  };
750          getkey;   if lextyp <> lexrparen   then error(erincrpar);
751          open(name)
752        end
753    end {  doinclude  };
754
755  { ******** }
756  {  doindex - process $index(expression)   }
757  { ******** }
758   procedure doindex;
759
760    var
761                    i: lnrng;
762
763    begin
764      over(ctop, maxcons);   ctop := ctop + 1;   getkey;
765      if lextyp = lexrparen
766      then with ctab[ctop] do begin ctyp := tin;   ci := 0 end
767      else expression;
768      if lextyp <> lexrparen   then error(erindrpar)
769      else
770        begin
771          pushback;
772          with ctab[ctop] do
773            if not (ctyp in [terr, tin])   then error(erindxtyp)
774            else
775              if ctyp = tin
776              then
777                begin
778                  index := index + 1;   ci := ci + index;   convrt;
779                  over(mtop, maxcalls);   mtop := mtop + 1;
780                  with mstack[mtop] do
781                    begin
782                      margs := nil;   mnext := atop;   mlast := atop - 1;
783                      matop := atop;
784                      for i := lexlen downto 1 do
785                        begin
786                          mnext := mnext - 1;
787                          defstr[mnext] := lexstr[i]
788                        end;
789                      getch
790                    end
791                end
792        end;
793      ctop := ctop - 1
794    end {  doindex  };
795
796  { ******** }
797  {  dooptions - process $options(...)   }
798  { ******** }
799   procedure dooptions;
800
801    var
802                    i: integer;
803
804    begin
805      gettok;
806      while not (lextyp in [lexrparen, lexeof]) do
807        begin
808          if lextyp = lexalpha
809          then
810            if lexstr[1] in ['R', 'P', 'N', 'L', 'E']
811            then
812              case lexstr[1] of
813                'P', 'R':
814                  begin
815                    while not (ch in ['0' .. '9', ')']) do getch;
816                    i := 0;
817                    while ch in ['0' .. '9'] do
818                      begin i := 10 * i + ord(ch) - ord('0');   getch end;
819                    if (mincol <= i) and (i <= maxcol) then
820                      case lexstr[1] of
821                        'P': prcopt := i;
822                        'R': rcopt := i
823                      end { case  }
824                  end;
825                'N':
826                  if lexlen >= 3 then
827                    if lexstr[3] = 'L'   then listopt := false
828                    else if lexstr[3] = 'E'   then expropt := false;
829                'L': listopt := true;
830                'E': expropt := true
831              end
832            else error(eropttype)
833          else if lextyp <> lexcomma   then error(eropttype);
834          gettok
835        end
836    end {  dooptions  };
837
838  { ******** }
839  {  dosysmac - perform proper system macro  }
840  { ******** }
841   procedure dosysmac { d:drng };
842
843    begin
844      gettok;
845      if lextyp <> lexlparen   then error(ersyslpar)
846      else
847        case d of
848          sysinc: doinclude;
849          syscodeif: docodeif;
850          sysindex: doindex;
851          sysdefine: dodefine;
852          sysoption: dooptions
853        end
854    end {  dosysmac  };
855
856  { ******** }
857  {  error - write out error message  }
858  { ******** }
859   procedure error { err:errmsg };
860
861    var
862                    i: lnrng;
863
864    begin
865      need(2) {  make sure message fits on page  };
866      if listopt
867      then
868        begin
869          write(space, errflag);   for i := 1 to next - 1 do write(blank);
870          writeln(arrow)
871        end
872      else writeln(' AT LINE:', line: 2, ' (pascal line:', pline: 2, ')');
873      writeln(space, errprefix, err);   nerrors := nerrors + 1
874    end {  error  };
875
876  { ******** }
877  {  evalfns - evaluate a builtin function  }
878  { ******** }
879   procedure evalfns { f:fns };
880
```

```
881   begin
882     case f of
883       fabs: evalabs;
884       fatn: evalatn;
885       fchr: evalchr;
886       fcos: evalcos;
887       fexp: evalexp;
888       flen: evallen {  length of a string  };
889       fln: evalln;
890       fodd: evalodd;
891       ford: evalord;
892       frou: evalrou {  round  };
893       fsin: evalsin;
894       fsqr: evalsqr;
895       fstr: evalstr {  string of - make integer a string  };
896       ftru: evaltru {  truncate  }
897     end {  case  }
898   end {  evalfns  };
899
900  { ******** }
901  {  evalabs - evaluate the abs builtin function  }
902  { ******** }
903   procedure evalabs;
904
905   begin
906     with ctab[ctop] do
907       if typeis([tre, tin])
908       then case ctyp of
909           tin: ci := abs(ci);
910           tre: cr := abs(cr)
911         end
912       else experror(erabstype)
913   end {  evalabs  };
914
915  { ******** }
916  {  evalatn - evaluate the arctan builtin function  }
917  { ******** }
918   procedure evalatn;
919
920   begin
921     with ctab[ctop] do
922       if typeis([tre, tin])
923       then
924         case ctyp of
925           tin: begin cr := arctan(ci);   ctyp := tre end;
926           tre: cr := arctan(cr)
927         end { case }
928       else experror(eratntype)
929   end {  evalatn  };
930
931  { ******** }
932  {  evalchr - evaluate the chr builtin function  }
933  { ******** }
934   procedure evalchr;
935
936   var
937                 i: integer;
938
939   begin
940     with ctab[ctop] do
941       if ctyp = tin
942       then
943         begin
944           i := ci;   ctyp := tch;   over(cstop, atop);
945           cstop := cstop + 1;   clen := 1;   cstr[cstop] := chr(i);
946           cfirst := cstop
947         end
948       else experror(erchrtype)
949   end {  evalchr  };
950
951  { ******** }
952  {  evalcos - evaluate the cosine builtin function  }
953  { ******** }
954   procedure evalcos;
955
956   begin
957     with ctab[ctop] do
958       if typeis([tre, tin])
959       then
960         case ctyp of
961           tin: begin cr := cos(ci);   ctyp := tre end;
962           tre: cr := cos(cr)
963         end { case }
964       else experror(ercostype)
965   end {  evalcos  };
966
967  { ******** }
968  {  evalexp - evaluate the exp builtin function  }
969  { ******** }
970   procedure evalexp;
971
972   begin
973     with ctab[ctop] do
974       if typeis([tre, tin])
975       then
976         case ctyp of
977           tin: begin cr := exp(ci);   ctyp := tre end;
978           tre: cr := exp(cr)
979         end { case }
980       else experror(erextype)
981   end {  evalexp  };
982
983  { ******** }
984  {  evallen - evaluate the length builtin function  }
985  { ******** }
986   procedure evallen;
987
988   var
989                 i: integer;
990
991   begin
992     with ctab[ctop] do
993       if ctyp = tch
994       then
995         begin
996           i := clen;   cstop := cfirst - 1;   ctyp := tin;   ci := i
997         end
998       else experror(erlentype)
999   end {  evallen  };
1000
1001 { ******** }
1002 {  evalln - evaluate the ln builtin function  }
1003 { ******** }
1004  procedure evalln;
1005
1006   begin
1007     with ctab[ctop] do
1008       if typeis([tre, tin])
1009       then
1010         case ctyp of
1011           tin: begin cr := ln(ci);   ctyp := tre end;
1012           tre: cr := ln(cr)
1013         end { case }
1014       else experror(erlntype)
1015   end {  evalln  };
1016
1017 { ******** }
1018 {  evalodd - evaluate the odd builtin function  }
1019 { ******** }
1020  procedure evalodd;
1021
1022   var
1023                 i: integer;
1024
1025   begin
1026     with ctab[ctop] do
1027       if ctyp = tin
1028       then begin i := ci;   ctyp := tbl;   cb := odd(i) end
1029       else experror(eroddtype)
1030   end {  evavodd  };
1031
1032 { ******** }
1033 {  evalord - evaluate the ord builtin function  }
1034 { ******** }
1035  procedure evalord;
1036
1037   var
1038                 c: char;
1039
1040   begin
1041     with ctab[ctop] do
1042       if ctyp = tch
1043       then
1044         if clen = 1
1045         then begin c := cstr[cfirst];   ctyp := tin;   ci := ord(c) end
1046         else experror(erordarg)
1047       else experror(erordtype)
1048   end {  evalord  };
1049
1050 { ******** }
1051 {  evalrou - evaluate the round builtin function  }
1052 { ******** }
1053  procedure evalrou;
1054
1055   var
1056                 r: real;
1057
1058   begin
1059     with ctab[ctop] do
1060       if ctyp = tre
1061       then begin r := cr;   ctyp := tin;   ci := round(r) end
1062       else experror(erroutype)
1063   end {  evalrou  };
1064
1065 { ******** }
1066 {  evalsin - evaluate the sin builtin function  }
1067 { ******** }
1068  procedure evalsin;
1069
1070   begin
1071     with ctab[ctop] do
1072       if typeis([tre, tin])
1073       then
1074         case ctyp of
1075           tin: begin cr := sin(ci);   ctyp := tre end;
1076           tre: cr := sin(cr)
1077         end { case }
1078       else experror(ersintype)
1079   end {  evalsin  };
1080
1081 { ******** }
1082 {  evalsqr - evaluate the sqr builtin function  }
1083 { ******** }
1084  procedure evalsqr;
1085
1086   begin
1087     with ctab[ctop] do
1088       if typeis([tre, tin])
1089       then
1090         case ctyp of
1091           tin: ci := sqr(ci);
1092           tre: cr := sqr(cr)
1093         end {  case  }
1094       else experror(ersqrtype)
1095   end {  evalsqr  };
1096
1097 { ******** }
1098 {  evalstr - evaluate the stringof builtin function  }
1099 { ******** }
1100  procedure evalstr;
```

```
1101
1102    var
1103                    i: integer;
1104                    c: char;
1105                  sgn: boolean;
1106
1107    begin
1108      with ctab[ctop] do
1109        if ctyp <> tin   then experror(erstrtype)
1110        else
1111          begin
1112            i := ci;
1113            if i < 0   then begin sgn := true;   i := abs(i) end
1114            else sgn := false;
1115            over(cstop, atop);   cstop := cstop + 1;   ctyp := tch;
1116            cfirst := cstop;
1117            if i = 0   then begin clen := 1;   cstr[cstop] := zero end
1118            else
1119              begin
1120                clen := 0;
1121                while i > 0 do
1122                  begin
1123                    cstr[cstop] := chr(ord(zero) + (i mod 10));
1124                    i := i div 10;   over(cstop, atop);
1125                    cstop := cstop + 1;   clen := clen + 1
1126                  end;
1127                if sgn   then cstr[cstop] := minus
1128                else cstop := cstop - 1;
1129                for i := 0 to(clen - 1) div 2 do
1130                  begin
1131                    c := cstr[i + cfirst];
1132                    cstr[i + cfirst] := cstr[cfirst + clen - i - 1];
1133                    cstr[cfirst + clen - i - 1] := c
1134                  end
1135              end
1136          end
1137    end { evalstr };
1138
1139  { ******** }
1140  { evaltru - evaluate trunc builtin function }
1141  { ******** }
1142    procedure evaltru;
1143
1144    var
1145                    r: real;
1146
1147    begin
1148      with ctab[ctop] do
1149        if ctyp = tre
1150        then begin r := cr;   ctyp := tin;   ci := trunc(r) end
1151        else experror(ertrutype)
1152    end { evaltru };
1153
1154  { ******** }
1155  { experror - print error for expression and flush }
1156  { ******** }
1157    procedure experror { err:errmsg };
1158
1159    begin error(err);   ctab[ctop].ctyp := terr;   flush
1160    end { experror };
1161
1162  { ******** }
1163  { expression - parse expression; put value in ctab[ctop] }
1164  { ******** }
1165    procedure expression;
1166
1167    begin
1168      relate;
1169      if typeis([tch])
1170      then
1171        begin
1172          over(ctop, maxcons);   ctop := ctop + 1;
1173          while lextyp in [lexst, lexalpha] do
1174            begin
1175              relate;
1176              if typeis([tch])
1177              then with ctab[ctop - 1] do clen := clen + ctab[ctop].clen
1178              else if not typeis([terr])   then experror(erexptype)
1179            end;
1180          ctop := ctop - 1;
1181        end
1182    end { expression };
1183
1184  { ******** }
1185  { factor - recognize factor part of expression }
1186  { ******** }
1187    procedure factor;
1188
1189    var
1190                    op: lex;
1191
1192    begin
1193      if lextyp in [lexnot, lexsub]
1194      then
1195        begin
1196          op := lextyp;   getkey;   factor;
1197          with ctab[ctop] do
1198            if typeis([tbl]) and (op = lexnot)   then cb := not cb
1199            else
1200              if typeis([tin, tre]) and (op = lexsub)
1201              then
1202                case ctyp of
1203                  tin: ci := - ci;
1204                  tre: cr := - cr
1205                end { case }
1206              else
1207                if ctyp <> terr
1208                then begin ctyp := terr;   experror(erfactype) end
1209        end
1210      else
1211        if lextyp = lexlparen
1212        then
1213          begin
1214            getkey;   expression;
1215            if not typeis([terr]) then
1216              if lextyp <> lexrparen   then experror(erfacrpar)
1217              else getkey
1218          end
1219        else variable
1220    end { factor };
1221
1222  { ******** }
1223  { findcon - find previously defined constant }
1224  { ******** }
1225    procedure findcon { name:alfa; var found:boolean };
1226
1227    var
1228                    c: crng;
1229                    i: integer;
1230
1231    begin
1232      c := cvalid;   ctab[0].cname := name;
1233      while ctab[c].cname <> name do c := c - 1;
1234      if c > 0
1235      then
1236        begin
1237          ctab[ctop] := ctab[c];
1238          with ctab[ctop] do
1239            if ctyp = tch
1240            then
1241              begin
1242                over(cstop + clen, maxcstr);   cfirst := cstop + 1;
1243                for i := 0 to clen - 1 do
1244                  begin
1245                    cstop := cstop + 1;
1246                    cstr[cstop] := cstr[ctab[c].cfirst + i]
1247                  end
1248              end;
1249          found := true
1250        end
1251    end { findcon };
1252
1253  { ******** }
1254  { flookup - lookup function name and return type code }
1255  { ******** }
1256    procedure flookup { name:alfa; var fun: fns; var found:boolean };
1257
1258    var
1259                    f: fnrng;
1260
1261    begin
1262      funct[0].fnnme := name;   f := maxfns;
1263      while funct[f].fnnme <> name do f := f - 1;
1264      if f = 0   then found := false
1265      else begin found := true;   fun := funct[f].fntyp end
1266    end { flookup };
1267
1268  { ******** }
1269  { flush - flush to semicolon }
1270  { ******** }
1271    procedure flush;
1272
1273    begin while not (lextyp in [lexeof, lexsemi]) do getkey
1274    end { flush };
1275
1276  { ******** }
1277  { forcereal - force top two constants on stack to real }
1278  { ******** }
1279    procedure forcereal;
1280
1281    var
1282                    i: integer;
1283
1284    begin
1285      with ctab[ctop] do
1286        if ctyp = tin   then begin i := ci;   ctyp := tre;   cr := i end;
1287      with ctab[ctop - 1] do
1288        if ctyp = tin   then begin i := ci;   ctyp := tre;   cr := i end
1289    end { forcereal };
1290
1291  { ******** }
1292  { getactuals - get actual parameters for macro call }
1293  { ******** }
1294    procedure getactuals { f:fptr; var act:aptr };
1295
1296    begin
1297      if f = nil
1298      then { if no formals, then no actuals }
1299      else
1300        begin
1301          new(act);
1302          with act^, f^ do
1303            begin
1304              aform := fname;   alast := atop - 1;   getparm;
1305              afirst := atop;   if ch = comma   then getch;
1306              getactuals(fnext, anext)
1307            end
1308        end;
1309    end { getactuals };
1310
1311  { ******** }
1312  { getbody - get the body of a macro }
1313  { ******** }
1314    procedure getbody;
1315
1316    var
1317                  ctr: integer { left parenthesis counter };
1318
1319    begin
1320      if ch = rparen
```

```
    then
      with defs[dtop] do
        begin getch;   dlast := dstop;   dfirst := dstop + 1 end
    else
      begin
        ctr := 1;
        with defs[dtop] do
          begin
            while ctr > 0 do
              begin
                over(dstop, atop);   dstop := dstop + 1;
                defstr[dstop] := ch;   dlast := dstop;
                if ch = rparen   then ctr := ctr - 1
                else
                  if ch = lparen   then ctr := ctr + 1
                  else
                    if (ch = newline) and (ftop = 0) and eof(fstack[0].
                      ffile)
                    then begin error(erbodyeof);   goto 1 end;
                getch
              end;
            defstr[dlast] := blank {  replace trailing ")"  }
          end
      end
  end {  getbody  };

{ ******** }
{  getbsu - get basic syntatic unit, subst. macro calls  }
{ ******** }
 procedure getbsu;

  var
              name: alfa;
             found: boolean;

  begin
    gettok;
    while lextyp = lexmac do
      begin
        pack(lexstr, 1, name);   ckformal(name, found);
        if not found then
          begin
            ckmacro(name, found);
            if not found   then begin error(ermacdefn);   gettok end
          end;
      end
  end {  getbsu  };

{ ******** }
{  getcdparm - get "codeif" code and save it  }
{ ******** }
 procedure getcdparm;

  var
            ctr: integer;
              d: dsrng;

  begin
    d := dstop;   ctr := 0;
    while (ctr > 0) or (ch <> rparen) do
      begin
        over(d, atop);   d := d + 1;   defstr[d] := ch;
        if ch = lparen   then ctr := ctr + 1
        else if ch = rparen   then ctr := ctr - 1;
        getch
      end;
    if d > dstop then
      begin
        over(d, atop);   d := d + 1;   defstr[d] := blank;
        while d > dstop do
          begin
            atop := atop - 1;   defstr[atop] := defstr[d];   d := d - 1
          end
      end
  end {  getcdparm  };

{ ******** }
{  getch - get next character and place in ch  }
{ ******** }
 procedure getch;

  begin
    if mtop > 0 then
      while (mstack[mtop].mnext > mstack[mtop].mlast) and (mtop > 0) do
        begin atop := mstack[mtop].matop;   mtop := mtop - 1; end;
    if mtop > 0
    then
      with mstack[mtop] do
        begin ch := defstr[mnext];   mnext := mnext + 1 end
    else
      begin
        if next > last   then getline;   ch := inline[next];
        next := next + 1
      end
  end {  getch  };

{ ******** }
{  getformals - get formal parameter names  }
{ ******** }
 procedure getformals { var f:fptr };

  begin
    if lextyp <> lexalpha   then f := nil
    else
      begin
        new(f);   lexstr[0] := dollar;   pack(lexstr, 0, f^.fname);
        gettok;
        if lextyp = lexcomma
        then begin gettok;   getformals(f^.fnext) end
        else f^.fnext := nil
      end
  end {  getformals  };

{ ******** }
{  getkey - get token and classify language keywords  }
{ ******** }
 procedure getkey;

  var
              name: alfa {  name of constant  };
                 k: krng {  pointer to keywords  };

  begin
    getbsu;
    if lextyp = lexalpha
    then
      begin
        pack(lexstr, 1, name);   keywd[0].kname := name;   k := maxkeys;
        while keywd[k].kname <> name do k := k - 1;
        if k > 0   then lextyp := keywd[k].klex
      end
  end {  getkey  };

{ ******** }
{  getline - place input line in linline; set next, last  }
{ ******** }
 procedure getline;

  var
            incol: lnrng;
               i: integer;

  begin
    while eof(fstack[ftop].ffile) and (ftop > 0) do close;
    if eof(fstack[ftop].ffile)
    then begin next := 1;   last := 0;   inline[next] := newline end
    else
      with fstack[ftop] do
        begin
          line := line + 1;   fline := fline + 1;   incol := 1;
          if listopt
          then
            begin
              if linectr >= pagesize
              then begin linectr := 0;   newpg end;
              linectr := linectr + 1;   write(space, line: 4, '   ');
              for i := 1 to 7 do write(fname[i]);
              write(fline: 5, pline: 8, '   ');
              while (not eoln(ffile)) and (incol <= rcopt) and (ffile^
                 = blank) do
                begin get(ffile);   write(blank);   incol := incol + 1
                end;
              next := incol;
              inline[next] := newline {  in case of empty line  };
              while (not eoln(ffile)) and (incol <= rcopt) do
                begin
                  inline[incol] := ffile^;   incol := incol + 1;
                  write(ffile^);   get(ffile)
                end;
              last := incol - 1;
              if not eoln(ffile) then
                begin
                  write(break);
                  while not eoln(ffile) and (incol < maxcol) do
                    begin write(ffile^);   get(ffile) end
                end;
              writeln
            end
          else
            begin
              while (not eoln(ffile)) and (incol <= rcopt) and (ffile^
                 = blank) do
                begin get(ffile);   incol := incol + 1 end;
              next := incol;
              inline[next] := newline {  in case of empty line  };
              while (not eoln(ffile)) and (incol <= rcopt) do
                begin
                  inline[incol] := ffile^;   incol := incol + 1;
                  get(ffile)
                end;
              last := incol - 1
            end;
          readln(ffile);
          if last >= next
          then begin last := last + 1;   inline[last] := newline end
        end
  end {  getline  };

{ ******** }
{  getparm - get an actual parm and save  }
{ ******** }
 procedure getparm;

  var
            ctr: integer;
              d: dsrng;

  begin
    d := dstop;   ctr := 0;
    while (ctr > 0) or not (ch in [comma, rparen]) do
      begin
        over(d, atop);   d := d + 1;   defstr[d] := ch;
        if ch = lparen   then ctr := ctr + 1
        else if ch = rparen   then ctr := ctr - 1;
        getch
      end;
    if d > dstop
    then
      begin
        over(d, atop);   d := d + 1;   defstr[d] := blank;
```

```
      while d > dstop do
        begin { move parm to right }
          atop := atop - 1;  defstr[atop] := defstr[d];  d := d - 1
        end
    end
end { getparm };

{ ******** }
{ gettok - get a token; set lexstr, lexlen, lextyp }
{ ******** }
procedure gettok;

var
          i: integer;
        num: integer { value of octal number };

begin
  lexlen := 0;
  while lexlen = 0 do
    begin
      while ch = blank do getch;  lexlen := 1;  lextyp := lexother;
      lexstr[1] := ch;
      case ch of
        newline:
          if (ftop = 0) and eof(fstack[ftop].ffile)
          then lextyp := lexeof
          else begin getch;  lexlen := 0 end;
        'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L',
          'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
          'Y', 'Z':
          begin
            getch;  lextyp := lexalpha;
            while ch in ['A' .. 'Z', '0' .. '9'] do
              begin
                lexlen := lexlen + 1;  lexstr[lexlen] := ch;  getch
              end;
            if lexlen > 10  then lexlen := 10;
            for i := lexlen + 1 to 10 do lexstr[i] := blank
          end;
        '0', '1', '2', '3', '4', '5', '6', '7', '8', '9':
          begin
            getch;  lextyp := lexint;
            while ch in ['0' .. '9'] do
              begin
                lexlen := lexlen + 1;  lexstr[lexlen] := ch;  getch
              end;
            if ch = letterb
            then
              begin { octal }
                getch;  num := 0;
                for i := 1 to lexlen do
                  if lexstr[i] in ['0' .. '7']
                  then num := 8 * num + ord(lexstr[i]) - ord(zero)
                  else begin num := 8 * num;  error(eroctdig) end;
                over(ctop, maxcons);  ctop := ctop + 1;
                with ctab[ctop] do begin ctyp := tin;  ci := num end;
                convrt;  ctop := ctop - 1
              end
            else
              begin
                if ch = period
                then
                  begin
                    getch;
                    if ch = period  then pushback
                    else
                      begin
                        lextyp := lexreal;  lexlen := lexlen + 1;
                        lexstr[lexlen] := period;
                        while ch in ['0' .. '9'] do
                          begin
                            lexlen := lexlen + 1;
                            lexstr[lexlen] := ch;  getch
                          end
                      end
                  end;
                if ch = lettere
                then
                  begin
                    lextyp := lexreal;  lexlen := lexlen + 1;
                    lexstr[lexlen] := ch;  getch;
                    if ch in [plus, minus] then
                      begin
                        lexlen := lexlen + 1;  lexstr[lexlen] := ch;
                        getch
                      end;
                    while ch in ['0' .. '9'] do
                      begin
                        lexlen := lexlen + 1;  lexstr[lexlen] := ch;
                        getch
                      end
                  end
              end
          end;
        '+': begin lextyp := lexadd;  getch end;
        '-': begin lextyp := lexsub;  getch end;
        '*': begin lextyp := lexmult;  getch end;
        '/': begin lextyp := lexdvd;  getch end;
        '(':
          begin
            getch;
            if ch <> star  then lextyp := lexlparen
            else
              begin
                getch;
                if ch = dollar
                then
                  begin
                    lexlen := 3;  unpack('(*$', lexstr, 1);
                    repeat
                      repeat
                        getch;  lexlen := lexlen + 1;
                        lexstr[lexlen] := ch
                      until ch = star;
                      getch;  lexlen := lexlen + 1;
                      lexstr[lexlen] := ch
                    until ch = rparen;
                    getch
                  end
                else
                  begin
                    lexlen := 0;
                    repeat while ch <> star do getch;  getch
                    until ch = rparen;
                    getch
                  end
              end
          end;
        ')': begin lextyp := lexrparen;  getch end;
        '$':
          begin
            getch;
            if not (ch in ['A' .. 'Z'])
            then begin error(ermacname);  lexlen := 0 end
            else
              begin
                lextyp := lexmac;
                while ch in ['A' .. 'Z', '0' .. '9'] do
                  begin
                    lexlen := lexlen + 1;  lexstr[lexlen] := ch;
                    getch
                  end;
                if lexlen > 10  then lexlen := 10;
                for i := lexlen + 1 to 10 do lexstr[i] := blank
              end
          end;
        '=': begin lextyp := lexeq;  getch end;
        ',': begin lextyp := lexcomma;  getch end;
        '.':
          begin
            getch;
            if ch = period then
              begin lexstr[2] := period;  lexlen := 2;  getch end
          end;
        '''':
          begin { extract string including all quotes }
            lexlen := 0;
            repeat
              over(lexlen, maxline);  lexlen := lexlen + 1;
              lexstr[lexlen] := ch;
              repeat
                getch;
                if ch = newline then
                  begin
                    error(erlongstr);  pushback;
                    ch := quote { supply missing quote }
                  end;
                over(lexlen, maxline);  lexlen := lexlen + 1;
                lexstr[lexlen] := ch
              until lexstr[lexlen] = quote;
              getch
            until ch <> quote;
            lextyp := lexst
          end;
        ':':
          begin
            getch;
            if ch = equal
            then begin lexlen := 2;  lexstr[2] := equal;  getch end
          end;
        '#':
          begin
            lextyp := lexne;  unpack('<>', lexstr, 1);  lexlen := 2;
            getch
          end;
        '!':
          begin
            lextyp := lexor;  unpack('OR', lexstr, 1);  lexlen := 2;
            getch
          end;
        '&':
          begin
            lextyp := lexand;  unpack('AND', lexstr, 1);
            lexlen := 3;  getch
          end;
        '<':
          begin
            getch;
            if ch = equal
            then
              begin
                lexlen := 2;  lexstr[2] := equal;  lextyp := lexle;
                getch
              end
            else
              if ch = greater
              then
                begin
                  lexlen := 2;  lexstr[2] := greater;
                  lextyp := lexne;  getch
                end
              else lextyp := lexlt
          end;
        '>':
          begin
            getch;
            if ch = equal
            then
              begin
                lexlen := 2;  lextyp := lexge;  lexstr[2] := equal;
```

```
                getch
              end
            else lextyp := lexgt
          end;
        '@':
          begin
            lextyp := lexle;   unpack('<=', lexstr, 1);   lexlen := 2;
            getch
          end;
        '\':
          begin
            lextyp := lexge;   unpack('>=', lexstr, 1);   lexlen := 2;
            getch
          end;
        '^':
          begin
            lextyp := lexnot;   unpack('NOT', lexstr, 1);
            lexlen := 3;   getch
          end;
        ';': begin lextyp := lexsemi;   getch end;
        '[', ']', '^', '_', '?': getch {  all other characters  }
      end {  case  }
    end
end {  gettok  };

{ ******** }
{  initialize - perform all necessary initialization  }
{ ******** }
procedure initialize;

 var
                i: integer;

 begin
   timein := clock;
   with ctab[1] do
     begin
       cname := 'MM        ';   ctyp := tch;   clen := 2;
       cfirst := 1
     end;
   with ctab[2] do
     begin
       cname := 'DD        ';   ctyp := tch;   clen := 2;
       cfirst := 4
     end;
   with ctab[3] do
     begin
       cname := 'YY        ';   ctyp := tch;   clen := 2;
       cfirst := 7
     end;
   with ctab[4] do
     begin
       cname := 'TIME      ';   ctyp := tch;   clen := 8;
       cfirst := 9
     end;
   with ctab[5] do
     begin
       cname := 'DATE      ';   ctyp := tch;   clen := 8;
       cfirst := 1
     end;
   with ctab[6] do
     begin cname := 'TRUE      ';   ctyp := tbl;   cb := true end;
   with ctab[7] do
     begin cname := 'FALSE     ';   ctyp := tbl;   cb := false end;
   with ctab[8] do
     begin cname := 'MAXINT    ';   ctyp := tre;   cr := maxint end;
   with ctab[9] do
     begin cname := 'MININT    ';   ctyp := tre;   cr := - maxint end;
   ctop := ndefconst {  number of predefined constants  };
   cvalid := ndefconst;
   timedate {  put mm/dd/yyhh:mm:ss into cstr[1..16]  };
  {  keywords are in order of decreasing frequency of access  }
   with keywd[16] do begin kname := 'AND       ';   klex := lexand end;
   with keywd[20] do begin kname := 'BEGIN     ';   klex := lexbeg end;
   with keywd[14] do begin kname := 'CASE      ';   klex := lexcas end;
   with keywd[10] do begin kname := 'CONST     ';   klex := lexcon end;
   with keywd[11] do begin kname := 'DIV       ';   klex := lexdiv end;
   with keywd[21] do begin kname := 'END       ';   klex := lexend end;
   with keywd[ 8] do begin kname := 'EXTERN    ';   klex := lexfwd end;
   with keywd[ 2] do begin kname := 'FORTRAN   ';   klex := lexfwd end;
   with keywd[15] do begin kname := 'FORWARD   ';   klex := lexfwd end;
   with keywd[ 9] do begin kname := 'FUNCTION  ';   klex := lexfun end;
   with keywd[ 4] do begin kname := 'MAX       ';   klex := lexmax end;
   with keywd[ 3] do begin kname := 'MCONST    ';  klex := lexmcon end;
   with keywd[ 5] do begin kname := 'MIN       ';   klex := lexmin end;
   with keywd[ 6] do begin kname := 'MOD       ';   klex := lexmod end;
   with keywd[17] do begin kname := 'NOT       ';   klex := lexnot end;
   with keywd[12] do begin kname := 'OR        ';   klex := lexor end;
   with keywd[19] do
     begin kname := 'PROCEDURE ';   klex := lexproc end;
   with keywd[13] do begin kname := 'RECORD    ';   klex := lexrec end;
   with keywd[ 1] do begin kname := 'RUN       ';   klex := lexfwd end;
   with keywd[ 7] do begin kname := 'TYPE      ';   klex := lextpe end;
   with keywd[18] do begin kname := 'VAR       ';   klex := lexvar end;
   mtop := 0;   dstop := 0;   defs[sysinc].dname := '$INCLUDE  ';
   defs[sysdefine].dname := '$DEFINE   ';
   defs[sysindex].dname := '$INDEX    ';
   defs[sysoption].dname := '$OPTIONS  ';
   defs[syscodeif].dname := '$CODEIF   ';   dtop := nsysmac;
   atop := maxdefstr {  actuals in rhs of dstr  };
   with funct[ 1] do begin fnnme := 'ABS       ';   fntyp := fabs end;
   with funct[ 2] do begin fnnme := 'ARCTAN    ';   fntyp := fatn end;
   with funct[ 3] do begin fnnme := 'CHR       ';   fntyp := fchr end;
   with funct[ 4] do begin fnnme := 'COS       ';   fntyp := fcos end;
   with funct[ 5] do begin fnnme := 'EXP:      ';   fntyp := fexp end;
   with funct[ 6] do begin fnnme := 'LENGTH    ';   fntyp := flen end;
   with funct[ 7] do begin fnnme := 'LN        ';   fntyp := fln end;
   with funct[ 8] do begin fnnme := 'ODD       ';   fntyp := fodd end;
   with funct[ 9] do begin fnnme := 'ORD       ';   fntyp := ford end;
   with funct[10] do begin fnnme := 'ROUND     ';   fntyp := frou end;
   with funct[11] do begin fnnme := 'SIN       ';   fntyp := fsin end;
   with funct[12] do begin fnnme := 'SQR       ';   fntyp := fsqr end;
   with funct[13] do begin fnnme := 'STRINGOF  ';   fntyp := fstr end;
   with funct[14] do begin fnnme := 'TRUNC     ';   fntyp := ftru end;
   line := 0 {  last line number for listing  };
   pline := 1 {  next, not last, pascal line number  };
   rewrite(psource);   rcopt := defrc;   prcopt := defprc;
   listopt := deflist;
   expropt := defexpr {  parse const expressions  };
   outpos := 0 {  last output position used  };
   lastlex := lexeof {  last token type output  };   nerrors := 0;
   index := 0;
   confl := [lexalpha, lexreal, lexint, lexand, lexor, lexnot, lexmin,
      lexmax, lexdiv, lexmod, lexbeg, lexcas, lexend, lexrec, lexfun,
      lexproc, lexcon, lextpe, lexvar];
   linectr := pagesize {  force newpage on listing  };
   ftop := - 1 {  no open files  };   open(inname);
   fstack[0].fname := inlname
 end {  initialize  };

{ ******** }
{  need - need l lines: start new page if necessary  }
{ ******** }
procedure need { l:pgrng };

 begin
   if (linectr + l) > pagesize   then begin linectr := l;   newpg end
   else linectr := linectr + l
 end { need };

{ ******** }
{ newpg - skip to a new page and print the heading }
{********}
procedure newpg;

 begin
   writeln(newpage, title1, title1a, dte: 9, title1b, tme: 9);
   writeln(double, title2);   writeln(space, title3);
   write(space, title4);   writeln(title5, title6)
 end { newpg };
{ newpg  }

{ ******** }
{  open - open an included file  }
{ ******** }
procedure open { name:alfa };

 var
                f: flrng;

 begin
   over(ftop, maxfiles);   fstack[ftop + 1].fname := name;   f := 0;
   while fstack[f].fname <> name do f := f + 1;
   if f <= ftop   then error(eropen)
   else
     begin
       ftop := ftop + 1;
       with fstack[ftop] do
         begin
           fname := name;
           { ffile must be opened with name fname  }
           reset(ffile);   fline := 0;   last := 0;   next := 1;
           inline[next] := newline;   mtop := 0;   getch
         end
     end
 end {  open  };

{ ******** }
{  over - abort on overflow  }
{ ******** }
procedure over { i:integer; maxval:integer };

 begin if i >= maxval   then begin error(erover);   goto 1 end
 end {  over  };

{ ******** }
{  parse - parse the input program  }
{ ******** }
procedure parse { top:crng; tok:lex };

 begin
   getkey;
   while not (lextyp in [lexeof, lexend, lexfwd]) do
     if lextyp in [lexrec, lexfun, lexproc, lexcon, lexmcon, lexbeg,
        lexcas]
     then
       case lextyp of
         lexbeg:
           begin
             puttok;
             if tok in [lexproc, lexfun]
             then begin tok := lexbeg;   getkey end
             else parse(ctop, lexbeg)
           end;
         lexcas:
           begin
             puttok;
             if tok = lexrec   then getkey   else parse(ctop, lexcas)
           end;
         lexcon:
           begin puttok;   if expropt   then parsecon   else getkey
           end;
         lexfun: begin puttok;   scanheader;   parse(ctop, lexfun) end;
         lexmcon: parsemcon;
         lexproc:
           begin puttok;   scanheader;   parse(ctop, lexproc) end;
         lexrec: begin puttok;   parse(ctop, lextyp) end
       end { case }
     else begin puttok;   getkey end;
   puttok;
```

```
1981      if (lextyp = lexeof) and (tok <> lexeof)
1982      then begin error(erparseof);   goto 1 end
1983      else
1984        if (lextyp = lexend) and not (tok in [lexbeg, lexcas, lexrec])
1985        then error(erparsend)
1986        else
1987          if (lextyp = lexfwd) and not (tok in [lexproc, lexfun])
1988          then error(erparsfwd);
1989      if lextyp <> lexeof   then getkey;   ctop := top;   cvalid := top
1990    end {  parse  };
1991
1992  { ******** }
1993  {  parsecon - parse a constant declaration with expression  }
1994  { ******** }
1995   procedure parsecon;
1996
1997    var
1998              savtyp: lex;
1999              savstr: strng;
2000              savlen: lnrng;
2001              svalid: boolean;
2002             consnam: alfa;
2003
2004    begin
2005      getkey;
2006      while lextyp = lexalpha do
2007        begin
2008          puttok;   over(ctop, maxcons);   ctop := ctop + 1;
2009          pack(lexstr, 1, consnam);   getkey;
2010          if lextyp <> lexeq
2011          then
2012            begin
2013              error(erparscon);   ctab[ctop].ctyp := terr;   flush;
2014              getkey
2015            end
2016          else
2017            begin
2018              puttok;   getkey;   while ch = blank do getch;
2019              if (ch = semi) and (lextyp in [lexint, lexreal, lexother])
2020              then
2021                begin
2022                  savstr := lexstr;   savlen := lexlen;
2023                  savtyp := lextyp;   svalid := true
2024                end
2025              else svalid := false;
2026              expression;
2027              if (lextyp <> lexsemi) and (not typeis([terr])) then
2028                begin experror(erpconsyn);   ctab[ctop].ctyp := terr end;
2029              if ctab[ctop].ctyp <> terr
2030              then
2031                begin
2032                  if svalid
2033                  then
2034                    begin
2035                      lexstr := savstr;   lextyp := savtyp;
2036                      lexlen := savlen
2037                    end
2038                  else convrt;
2039                  puttok;   lextyp := lexsemi;   lexstr[1] := semi;
2040                  lexlen := 1;   puttok;   ctab[ctop].cname := consnam;
2041                  cvalid := ctop
2042                end
2043              else
2044                begin
2045                  lexstr[1] := zero;   lexstr[2] := semi;
2046                  lextyp := lexst;   lexlen := 2;   puttok
2047                end
2048            end;
2049          if ctab[ctop].ctyp in [terr, tot]   then ctop := ctop - 1;
2050          getkey
2051        end
2052    end {  parsecon  };
2053
2054  { ******** }
2055  { parsemcon - parse an internal constant declaration with expression }
2056  { ******** }
2057
2058   procedure parsemcon;
2059
2060    var
2061             consnam: alfa;
2062
2063    begin
2064      getkey;
2065      while lextyp = lexalpha do
2066        begin
2067          over(ctop, maxcons);   ctop := ctop + 1;
2068          pack(lexstr, 1, consnam);   getkey;
2069          if lextyp <> lexeq
2070          then
2071            begin
2072              error(erparsmcon);   ctab[ctop].ctyp := terr;   flush;
2073              getkey
2074            end
2075          else
2076            begin
2077              getkey;   while ch = blank do getch;   expression;
2078              if (lextyp <> lexsemi) and (not typeis([terr])) then
2079                begin experror(ermconsyn);   ctab[ctop].ctyp := terr end;
2080              if ctab[ctop].ctyp <> terr then
2081                begin ctab[ctop].cname := consnam;   cvalid := ctop end
2082            end;
2083          if ctab[ctop].ctyp in [terr, tot]   then ctop := ctop - 1;
2084          getkey
2085        end
2086    end {  parsemcon  };
2087
2088  { ******** }
2089  {  pushback - push character back onto input  }
2090  { ******** }
2091   procedure pushback;
2092
2093    begin
2094      if mtop > 0   then with mstack[mtop] do mnext := mnext - 1
2095      else next := next - 1
2096    end {  pushback  };
2097
2098  { ******** }
2099  {  puttok - put out a token for pascal using cols 1-prc  }
2100  { ******** }
2101   procedure puttok;
2102
2103    var
2104                  i: lnrng;
2105
2106    begin
2107      if (lastlex in confl) and (lextyp in confl) then
2108        begin
2109          write(psource, blank) {  space needed between tokens  };
2110          outpos := outpos + 1
2111        end;
2112      if lextyp = lexeof   then begin writeln(psource);   outpos := 0 end
2113      else
2114        begin
2115          if (outpos + lexlen) > prcopt
2116          then
2117            begin
2118              pline := pline + 1;   writeln(psource);   outpos := 0;
2119              if lexlen > prcopt
2120              then begin error(erputtok);   lexlen := prcopt end
2121            end;
2122          for i := 1 to lexlen do write(psource, lexstr[i]);
2123          outpos := outpos + lexlen;   lastlex := lextyp
2124        end
2125    end {  puttok  };
2126
2127  { ******** }
2128  {  relate - parse subexpression with rel. ops  }
2129  { ******** }
2130   procedure relate;
2131
2132    var
2133                 op: lex;
2134                  i: integer;
2135                  r: real;
2136                 c1,
2137                 c2: csrng;
2138
2139    begin
2140      arith;
2141      while (lextyp in [lexlt .. lexne]) and (not typeis([terr])) do
2142        begin
2143          over(ctop, maxcons);   ctop := ctop + 1;   op := lextyp;
2144          getkey;   arith;
2145          if typesmatch
2146          then
2147            with ctab[ctop - 1] do
2148              case ctyp of
2149                tin:
2150                  begin
2151                    i := ci;   ctyp := tbl;
2152                    case op of
2153                      lexlt: cb := i < ctab[ctop].ci;
2154                      lexle: cb := i <= ctab[ctop].ci;
2155                      lexeq: cb := i = ctab[ctop].ci;
2156                      lexge: cb := i >= ctab[ctop].ci;
2157                      lexgt: cb := i > ctab[ctop].ci;
2158                      lexne: cb := i <> ctab[ctop].ci
2159                    end {  case  }
2160                  end;
2161                tre:
2162                  begin
2163                    r := cr;   ctyp := tbl;
2164                    case op of
2165                      lexlt: cb := r < ctab[ctop].cr;
2166                      lexle: cb := r <= ctab[ctop].cr;
2167                      lexeq: cb := r = ctab[ctop].cr;
2168                      lexge: cb := r >= ctab[ctop].cr;
2169                      lexgt: cb := r > ctab[ctop].cr;
2170                      lexne: cb := r <> ctab[ctop].cr
2171                    end {  case  }
2172                  end;
2173                tbl:
2174                  case op of
2175                    lexlt: cb := cb < ctab[ctop].cb;
2176                    lexle: cb := cb <= ctab[ctop].cb;
2177                    lexeq: cb := cb = ctab[ctop].cb;
2178                    lexge: cb := cb >= ctab[ctop].cb;
2179                    lexgt: cb := cb > ctab[ctop].cb;
2180                    lexne: cb := cb <> ctab[ctop].cb
2181                  end;
2182                tot: begin experror(errelatyp);   ctyp := terr end;
2183                tch:
2184                  begin
2185                    c1 := cfirst;   c2 := ctab[ctop].cfirst;   i := 1;
2186                    while (i < clen) and (cstr[c1] = cstr[c2]) do
2187                      i := i + 1;
2188                    cstop := cstop - clen - ctab[ctop].clen;
2189                    ctyp := tbl;
2190                    case op of
2191                      lexlt: cb := cstr[c1] < cstr[c2];
2192                      lexle: cb := cstr[c1] <= cstr[c2];
2193                      lexeq: cb := cstr[c1] = cstr[c2];
2194                      lexge: cb := cstr[c1] >= cstr[c2];
2195                      lexgt: cb := cstr[c1] > cstr[c2];
2196                      lexne: cb := cstr[c1] <> cstr[c2]
2197                    end {  case  }
2198                  end
2199              end {  case  }
2200        else
```

```
2201          if ctab[ctop].ctyp <> terr
2202          then begin experror(errelconf);   ctab[ctop].ctyp := terr end;
2203        ctop := ctop - 1
2204      end
2205  end { relate };
2206
2207 { ******** }
2208 {  scanheader - scan procedure or function heading  }
2209 { ******** }
2210  procedure scanheader;
2211
2212  var
2213                ctr: integer;
2214
2215  begin
2216    getkey {  get name  };   puttok {   get name   };
2217    getkey {  get paren if parameters  };
2218    if lextyp <> lexlparen   then puttok
2219    else
2220      begin
2221        ctr := 1;   puttok;
2222        repeat
2223          getkey;   if lextyp = lexlparen   then ctr := ctr + 1;
2224          if lextyp = lexrparen   then ctr := ctr - 1;   puttok
2225        until ctr = 0
2226      end
2227  end { scanheader  };
2228
2229 { ******** }
2230 {  term - process multiplication ops in expression  }
2231 { ******** }
2232  procedure term;
2233
2234  var
2235                op: lex;
2236
2237  begin
2238    factor;
2239    if (lextyp in [lexand .. lexmod]) and (not typeis([terr]))
2240    then
2241      if (typeis([tbl]) and (lextyp = lexand)) or (typeis([tre]) and (
2242        lextyp in [lexmult .. lexmax])) or (typeis([tin]) and (lextyp
2243        in [lexmult .. lexmod]))
2244      then
2245        while lextyp in [lexand .. lexmod] do
2246          begin
2247            ctop := ctop + 1;   op := lextyp;   getkey;   factor;
2248            with ctab[ctop - 1] do
2249              if (op = lexand) and (ctyp = tbl)
2250              then cb := cb and ctab[ctop].cb
2251              else
2252                if (op in [lexdiv .. lexmod]) and (ctyp = tin)
2253                then
2254                  case op of
2255                    lexdiv: ci := ci div ctab[ctop].ci;
2256                    lexmod: ci := ci mod ctab[ctop].ci
2257                  end {  case  }
2258                else
2259                  if (op in [lexmult .. lexmax]) and typeis([tin, tre])
2260                  then
2261                    begin
2262                      if (ctyp = tin) and typeis([tin]) and (op <>
2263                        lexdvd)
2264                      then
2265                        case op of
2266                          lexmult: ci := ci * ctab[ctop].ci;
2267                          lexmin:
2268                            if ctab[ctop].ci < ci
2269                            then ci := ctab[ctop].ci;
2270                          lexmax:
2271                            if ctab[ctop].ci > ci
2272                            then ci := ctab[ctop].ci
2273                        end {  case  }
2274                      else
2275                        begin
2276                          forcereal;
2277                          case op of
2278                            lexmult: cr := cr * ctab[ctop].cr;
2279                            lexdvd: cr := cr / ctab[ctop].cr;
2280                            lexmin:
2281                              if ctab[ctop].cr < cr
2282                              then cr := ctab[ctop].cr;
2283                            lexmax:
2284                              if ctab[ctop].cr > cr
2285                              then cr := ctab[ctop].cr
2286                          end {  case  }
2287                        end
2288                    end
2289                  else
2290                    if ctab[ctop].ctyp <> terr
2291                    then experror(ertermtyp);
2292            ctop := ctop - 1
2293          end
2294      else error(ertermtyp)
2295  end {  term  };
2296
2297 { ******** }
2298 {  terminate - print statistics and close files  }
2299 { ******** }
2300  procedure terminate;
2301
2302  var
2303            ratio: real {  lines/sec ratio  };
2304
2305  begin
2306    if outpos > 0   then writeln(psource);
2307    if nerrors > 0 then
2308      begin
2309        need(2);
2310        writeln(double, '---> there were ', nerrors: 1,
2311           ' errors detected by map');
2312      end;
2313    tottme := clock - timein;
2314    if tottme = 0   then ratio := 0.0
2315    else ratio := 1000 * line / tottme;
2316    need(2);
2317    writeln(double, '---> end run: ', line: 5, ' input lines,', pline: 6
2318      , ' output lines,', tottme: 7, ' MS (', ratio: 8: 4,
2319      ' lines/sec)');
2320  end {  terminate  };
2321
2322 { ******** }
2323 {  timedate - get time and date and store in cstr  }
2324 { ******** }
2325  procedure timedate;
2326
2327   begin {  get time and date from system and make  }
2328 {  cstr[1..16] mm/dd/yyhh:mm:ss                    }
2329 {                                                  }
2330 {  global variables tme and dte should be         }
2331 {  set to time and date for the listing            }
2332 {  temporary time and date                         }
2333     unpack('MM/DD/YYHH:MM:SS', cstr, 1);   tme := '*TIME*    ';
2334     dte := '*TODAY*   '
2335   end {  timedate  };
2336
2337 { ******** }
2338 {  typeis - return true if type of top of stack is in set  }
2339 { ******** }
2340  function typeis { c:cset):boolean };
2341
2342   begin typeis := ctab[ctop].ctyp in c end {  typeis  };
2343
2344 { ******** }
2345 {  typesmatch - return true if types of top operands compatible  }
2346 { ******** }
2347  function typesmatch { :boolean };
2348
2349   begin
2350     typesmatch := false;
2351     with ctab[ctop - 1] do
2352       if ctyp = ctab[ctop].ctyp then
2353         if ctyp <> tch   then typesmatch := true
2354         else if clen = ctab[ctop].clen   then typesmatch := true
2355  end {  typesmatch  };
2356
2357 { ******** }
2358 {  variable - recognize variable in expression  }
2359 { ******** }
2360  procedure variable;
2361
2362   var
2363               name: alfa;
2364             found: boolean;
2365               fun: fns;
2366
2367   begin
2368     if not (lextyp in [lexalpha, lexint, lexreal, lexst])
2369     then begin experror(ervalexp);   ctab[ctop].ctyp := terr end
2370     else
2371       case lextyp of
2372         lexint: begin convrti;   getkey end;
2373         lexreal: begin convrtr;   getkey end;
2374         lexst: begin convrts;   getkey end;
2375         lexalpha:
2376           begin
2377             pack(lexstr, 1, name);   getkey;   found := false;
2378             if lextyp <> lexlparen
2379             then
2380               begin
2381                 findcon(name, found);
2382                 if not found then
2383                   with ctab[ctop] do
2384                     begin ctyp := tot;   co := name end
2385               end
2386             else
2387               begin
2388                 flookup(name, fun, found) {  function call  };
2389                 if not found   then experror(ervarfnct)
2390                 else
2391                   begin
2392                     getkey;   expression;
2393                     if lextyp <> lexrparen   then experror(ervarrpar)
2394                     else begin getkey;   evalfns(fun) end
2395                   end
2396               end
2397           end
2398       end {  case  }
2399   end {  variable  };
2400  begin { map }
2401   initialize;   parse(ctop, lexeof);
2402 1: terminate end.
```

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

```
program Xref(input, output, tty) {    N. Wirth  10.2.76      };

{ Cross Reference generator for Pascal programs              }
{  quadratic quotient hash method                            }
{  revised by R.J.Cichelli 16-Feb-79                         }
{   include perfect hash function, ring data structures,     }
{           and clean up code.                               }
{  revised by J.P.McGrath 22-May-79                          }
{   predefined identifier processing                         }
{   modified quicksort algorithm                             }
{   command line processing by M.Q.Thompson                  }
{  revised by R.J.Cichelli  26-Nov-79                        }
{   string table processing and work-files                   }
{  Copyright 1979          Pascal Users Group                }
{    permission to copy - except for profit - granted        }
{
 * Purpose:
     This program cross references Pascal programs.
     It supports upper and lower case, long identifiers and
     long programs.

 * Authors:
     N. Wirth, R.J.Cichelli, M.Q.Thompson, J.P.McGrath.

 * Method:
     Quadratic quotient hash method with tagged, quick-sorted string
     table and perfect hash function reserved word and predefined
     identifier filters.  Overflow processing by multi-file merge-sort.

 * Description of parameters:
     DEC PDP 11 RSX protocol.
     PXR <output file>=<input file> [<options>]
     <options> ::=
       C-  captalize identifiers,
       D+  display program,
       P-  cross reference predefined identifiers,
       T-  terminal output (80 columns and ids. only),
       W=132 width of output.

 * Input:
     Pascal Program source.

 * Output:
     Listing and references.

 * Limitations:

 * Computer system:
     Program was run under Seved Torstendahl's DEC PDP 11 RSX Pascal.
     This compiler (version 4.5) doesn't support program parameters
     in full generality.  In this program implemementation specific
     code handles control card cracking and file variable and system
     file name associations.

 * Installation under RSX:

      DP1:XREF/-FP/MU,TI:/SH=DP1:XREF.ODL/MP
      TASK=...PXR
      LIBR=SYSRES:RO
      EXTSCT=$HEAP1:40000
      EXTSCT=$$FSR1:5140
      UNITS=6
      //

              ;ODL  (overlay description)
              .ROOT  R1-*(O1,O2)
       R1:    .FCTR  DP1:XREF/LB:XREF:PAGEHE-DP0:[1,1]PASLIB/LB
       O1:    .FCTR  DP1:XREF/LB:QUICKS
       O2:    .FCTR  DP1:XREF/LB:INITPE-O3-*(O21,O22)
       O21:   .FCTR  DP1:XREF/LB:INITCH
       O22:   .FCTR  DP1:XREF/LB:INITPR
       O3:    .FCTR  DP0:[1,1]PASLIB/LB:GCML
              .END
 }

 {$R-   no runtime testing   }
 {$W-   no warning messages  }

const
  quote = '''';
  lCurleyBra = '{';
  rCurleyBra = '}';
  HashTblSize = 997          {  size of hash table - prime   };
  MaxItems = 4000    {  arbitrary limit on incore references  };
  StgTblSize = 6000          {  string table size            };
  StgTblLimit = 5900         {  limit is size - 100          };
  NumOfReserved = 40         {  size of reserved word table  };
  NumOfPredefnd = 48         {  size of predefined id table  };
  keylength = 10             {  keylength                    };
  DigitsPerNumber = 6        {  no. of digits per number     };
  LinesPerPage = 57          {  lines/page                   };
  DefaultTerminalWidth = 80  {  terminal width               };
  DefaultLpWidth = 132       {  line printer width           };
  MaxLineNo = maxint         {  maximum line number          };

type
          text = file of char;
         index = 0 .. HashTblSize;
    StgTblIndx = 1 .. StgTblSize;
          alfa = packed array [1 .. keylength] of char;
       ItemPtr = ^item;
          word = record
                   keyindx,
                    keylen: StgTblIndx;
                   lastptr: ItemPtr
                 end;
          item = packed record
                   LineNumber: 0 .. MaxLineNo;
                         next: ItemPtr
                 end;
    LineBuffer = packed array [1 .. 80] of char;
       ChrType = (ucLetter, lcLetter, digit, other);
     FilStates = (inout, inwrk1, wrk1out, wrk1wrk2, wrk2out,
                  wrk2wrk1);

var
        charindx,
           idlen,
     HshTblIndx: integer;
          empty: alfa;
     identifier: alfa;
CurrentLineNumber: integer     {  current line number  };
    LinesOnPage: integer       {  number of lines on current page  };
  LineNosPerLine: integer      {  no. of line-numbers per line  };
      HashTable: array [index] of word {  hash table  };
       StgTable: packed array [StgTblIndx] of char
                                {  for storing identifiers  };
     FreeStgPtr: integer;
    FreeItemPtr: ItemPtr;
        ItemCnt: integer;
    ChrCatagory: array [char] of ChrType;
     ChrSortOrd: array [char] of integer;
ReservRepresentedBy,
PredefRepresentedBy: array [char {  'A' .. '9'  } ] of integer;
  LastLeadingChar,
               ch,
            rawch,
          fstchar,
          lstchar: char;
         reserved: array [1 .. NumOfReserved] of alfa;
       predefined: array [1 .. NumOfPredefnd] of alfa;
       LineLength: integer;
          cmlline: LineBuffer;
           cmllen: integer;
            today,
              now: packed array [1 .. 10] of char;
  OutputSection: (listing, idents);
     PageNumber: integer;
  DisplayIsActive,
     DoPredefined,
         terminal,
      AllCapitals: Boolean;
            state: FilStates;
        NextState: array [FilStates, Boolean] of FilStates;
       wrk2active: Boolean;
             wrk1,
             wrk2: text;

procedure Pageheader;

  var
                i: integer;
          IsNarrow: 0 .. 1;

  begin
    IsNarrow := 0;
    if not terminal
    then
      begin
        PageNumber := PageNumber + 1;   page(output);
        write(' CrossRef  -  ');
        case OutputSection of
          listing: write('Program Listing                        ');
          idents: write('Identifier Cross-Reference             ')
        end;
        write('    ', today, '  ', now: 8);
        if LineLength <= DefaultTerminalWidth
        then begin writeln;   write(' ');   IsNarrow := 1; end
        else write('    ');
        for i := 1 to cmllen do write(cmlline[i]);
        write(' ': (25 * IsNarrow + 40 - cmllen));
        writeln(' Page ', PageNumber: 3);   writeln;
      end;
    LinesOnPage := IsNarrow;
  end {  pageheader  };

function UpperCase(ch: char): char;

  begin {  This should work for both ASCII and EBCDIC.  }
    if ChrCatagory[ch] = lcLetter
    then UpperCase := chr(ord(ch) - ord('a') + ord('A'))
    else UpperCase := ch;
  end {  uppercase  };

function EqlStg(indx1, len1, indx2, len2: integer): Boolean;

  var
              disp,
            StopAt: integer;

  begin
    if len1 <> len2   then EqlStg := false
    else
      begin
        disp := 0;   StopAt := len1 - 1;
        while (disp < StopAt) and (StgTable[indx1 + disp] = StgTable[
           indx2 + disp]) do
          disp := disp + 1;
        EqlStg := StgTable[indx1 + disp] = StgTable[indx2 + disp]
      end
  end { eqlstg };

function LssStg(indx1, len1, indx2, len2: integer): Boolean;

  var
             StopAt,
               disp,
              point: integer;

  begin
```

```
221     if len1 < len2   then StopAt := len1 - 1   else StopAt := len2 - 1;
222     disp := 0;
223     while (StgTable[indx1 + disp] = StgTable[indx2 + disp]) and (disp <
224        StopAt) do
225       disp := disp + 1;
226     point := disp;
227     while (UpperCase(StgTable[indx1 + disp]) = UpperCase(StgTable[indx2
228        + disp])) and (disp < StopAt) do
229       disp := disp + 1;
230     if UpperCase(StgTable[indx1 + disp]) = UpperCase(StgTable[indx2 +
231       disp])
232     then
233       if len1 = len2
234       then
235         LssStg := ChrSortOrd[StgTable[indx1 + point]] < ChrSortOrd[
236             StgTable[indx2 + point]]
237       else LssStg := len1 < len2
238     else
239       LssStg := ChrSortOrd[StgTable[indx1 + disp]] < ChrSortOrd[StgTable
240           [indx2 + disp]];
241   end { lssstg };
242
243   {$Y+  new segment  }
244
245 procedure PrintTables(var infil, out: text);
246
247   var
248             tryindx,
249              trylen: integer  {  quick sort temporaries  };
250            SwapWord: word {  quicksort temporary  };
251            midpoint: integer;
252             TblIndx,
253           MoveToIndx: index;
254                    i: integer;
255       NumberCounter: integer;
256           CmpRefPtr,
257           CmpRefLen: integer;
258
259   procedure QuickSort(LowerBound, UpperBound: integer);
260
261     var
262           TmpLowerBnd,
263           TmpUpperBnd: integer;
264
265     begin
266       repeat
267         TmpLowerBnd := LowerBound;   TmpUpperBnd := UpperBound;
268         midpoint := (TmpLowerBnd + TmpUpperBnd) div 2;
269         tryindx := HashTable[midpoint].keyindx;
270         trylen := HashTable[midpoint].keylen;
271         repeat
272           while LssStg(HashTable[TmpLowerBnd].keyindx, HashTable[
273              TmpLowerBnd].keylen, tryindx, trylen) do
274             TmpLowerBnd := TmpLowerBnd + 1;
275           while LssStg(tryindx, trylen, HashTable[TmpUpperBnd].keyindx,
276              HashTable[TmpUpperBnd].keylen) do
277             TmpUpperBnd := TmpUpperBnd - 1;
278           if TmpLowerBnd <= TmpUpperBnd
279           then
280             begin
281               SwapWord := HashTable[TmpLowerBnd];
282               HashTable[TmpLowerBnd] := HashTable[TmpUpperBnd];
283               HashTable[TmpUpperBnd] := SwapWord;
284               TmpLowerBnd := TmpLowerBnd + 1;
285               TmpUpperBnd := TmpUpperBnd - 1
286             end
287         until TmpLowerBnd > TmpUpperBnd;
288         if TmpUpperBnd - LowerBound < UpperBound - TmpLowerBnd
289         then
290           begin
291             if LowerBound < TmpUpperBnd
292             then QuickSort(LowerBound, TmpUpperBnd);
293             LowerBound := TmpLowerBnd;
294           end
295         else
296           begin
297             if TmpLowerBnd < UpperBound
298             then QuickSort(TmpLowerBnd, UpperBound);
299             UpperBound := TmpUpperBnd;
300           end;
301       until UpperBound <= LowerBound;
302     end {  quicksort  };
303
304   procedure EndLine(achar: char);
305
306     begin
307       if OutputSection = idents
308       then
309         begin
310           writeln(out);   LinesOnPage := LinesOnPage + 1;
311           if LinesOnPage > LinesPerPage
312           then begin Pageheader;   LinesOnPage := 1 end;
313         end
314       else writeln(out, achar);
315     end { endline };
316
317   procedure PrintNumbers(aword: word);
318
319     var
320               LoopPtr,
321               TailPtr: ItemPtr;
322
323     begin
324       TailPtr := aword.lastptr;   LoopPtr := TailPtr^.next;
325       TailPtr := LoopPtr;
326       repeat
327         if NumberCounter = LineNosPerLine then
328           begin
329             NumberCounter := 0;   EndLine(',');
330             write(out, ' ': keylength + ord(OutputSection = idents));
331           end;
332         NumberCounter := NumberCounter + 1;
333         write(out, LoopPtr^.LineNumber: DigitsPerNumber);
334         LoopPtr := LoopPtr^.next
335       until LoopPtr = TailPtr;
336     { free ring  }
337       aword.lastptr^.next := FreeItemPtr;   FreeItemPtr := LoopPtr;
338       EndLine('.');
339     end {  printnumbers  };
340
341   procedure NextRef;
342
343     begin
344       if CmpRefLen > 0
345       then
346         begin
347           CmpRefLen := 0;
348           if not eof(infil) then
349             repeat
350               StgTable[CmpRefPtr + CmpRefLen] := infil^;
351               CmpRefLen := CmpRefLen + 1;   get(infil)
352             until (infil^ = ' ')
353         end
354     end { nextref };
355
356   procedure OutId(keyptr, lenkey: integer; SetUpForNos: Boolean);
357
358     var
359               chindx: integer;
360
361     begin
362       if OutputSection = idents
363       then
364         begin
365           if (LinesOnPage + 4) > LinesPerPage
366           then begin Pageheader;   LinesOnPage := 1 end
367           else
368             if LastLeadingChar <> UpperCase(StgTable[keyptr])
369             then EndLine(' ');
370           write(out, ' ');
371           LastLeadingChar := UpperCase(StgTable[keyptr]);
372         end;
373       for chindx := keyptr to keyptr + lenkey - 1 do
374         write(out, StgTable[chindx]);
375       if SetUpForNos
376       then
377         begin
378           if lenkey > keylength
379           then
380             begin
381               write(out, ' ': ((DigitsPerNumber - 1) - ((lenkey - (
382                  keylength + 1)) mod DigitsPerNumber)));
383               NumberCounter := ((lenkey - keylength) div DigitsPerNumber
384                  ) + 1;
385             end
386           else
387             begin
388               write(out, ' ': (keylength - lenkey));
389               NumberCounter := 0
390             end;
391         end
392     end {  outid  };
393
394   procedure CopyRef(AllOfIt: Boolean);
395
396     var
397               lastlen: integer;
398
399     procedure CopyLines;
400
401       var
402                 RefDone: Boolean;
403                  savech: char;
404
405       begin
406         lastlen := CmpRefLen;   RefDone := false;
407         repeat
408           repeat
409             write(out, infil^);   lastlen := lastlen + 1;   get(infil)
410           until (infil^ = '.') or (infil^ = ',') or eoln(infil);
411           savech := infil^;
412           if savech = '.'
413           then
414             begin
415               RefDone := true;
416               if not AllOfIt then
417                 begin
418                   savech := ' ';
419                   NumberCounter := ((lastlen - keylength) div
420                      DigitsPerNumber);
421                 end;
422             end
423           else lastlen := 0;
424           while not eoln(infil) do get(infil);
425           if eof(infil)
426           then begin CmpRefLen := 0;   RefDone := true; end
427           else get(infil);
428           if savech <> ' ' then
429             begin
430               EndLine(savech);
431               if not RefDone and (OutputSection = idents)
432               then write(out, ' ');
433             end;
434         until RefDone
435       end {  copy lines  };
436
437     begin {  copyref  }
438       OutId(CmpRefPtr, CmpRefLen, false);   CopyLines;
439     end {  copyref  };
440
```

```
441   procedure syncronize(aword: word);
442
443     var
444                 done: Boolean;
445
446     begin
447       done := false;
448       with aword do
449         repeat
450           if CmpRefLen = 0
451           then begin done := true;   OutId(keyindx, keylen, true); end
452           else
453             if LssStg(CmpRefPtr, CmpRefLen, keyindx, keylen)
454             then begin CopyRef(true);   NextRef end
455             else
456               if EqlStg(CmpRefPtr, CmpRefLen, keyindx, keylen)
457               then begin CopyRef(false);   NextRef;   done := true; end
458               else
459                 begin OutId(keyindx, keylen, true);   done := true; end;
460       until done;
461     end {  syncronize  };
462
463   begin {  printtables  }
464     LinesOnPage := LinesPerPage;   MoveToIndx := 0 {  compress table  };
465     for TblIndx := 0 to HashTblSize - 1 do
466       if HashTable[TblIndx].keyindx <> 0 then
467         begin
468           HashTable[MoveToIndx] := HashTable[TblIndx];
469           MoveToIndx := MoveToIndx + 1
470         end;
471     if MoveToIndx > 0   then QuickSort(0, MoveToIndx - 1);
472     LineNosPerLine := (LineLength - keylength - 1) div DigitsPerNumber;
473     CmpRefPtr := FreeStgPtr + idlen;   LastLeadingChar := ' ';
474     if state > inwrk1
475     then begin CmpRefLen := 1;   NextRef {  first reference  }; end
476     else CmpRefLen := 0;
477     if HashTable[0].keyindx <> 0 then
478       for TblIndx := 0 to MoveToIndx - 1 do
479         begin
480           syncronize(HashTable[TblIndx]);
481           PrintNumbers(HashTable[TblIndx]);
482         end;
483     while CmpRefLen <> 0 do begin CopyRef(true);   NextRef; end;
484   end {  printtables  };
485
486 procedure DumpTables;
487
488   var
489               chptr: integer;
490
491   begin
492     state := NextState[state, (OutputSection = idents)];
493     case state of
494       inout: PrintTables(input {  dummy  }, output);
495       inwrk1:
496         begin
497 {}        rewrite(wrk1, 'XRF1JNK.TMP;1',, 'DPO:');
498           PrintTables(input {  dummy  }, wrk1);
499         end;
500       wrk1out:
501         begin
502 {}        reset(wrk1, 'XRF1JNK.TMP;1',, 'DPO:');
503           PrintTables(wrk1, output);
504         end;
505       wrk1wrk2:
506         begin
507 {}        reset(wrk1, 'XRF1JNK.TMP;1',, 'DPO:');
508 {}        rewrite(wrk2, 'XRF2JNK.TMP;1',, 'DPO:');
509           wrk2active := true;   PrintTables(wrk1, wrk2);
510         end;
511       wrk2out:
512         begin
513 {}        reset(wrk2, 'XRF2JNK.TMP;1',, 'DPO:');
514           PrintTables(wrk2, output)
515         end;
516       wrk2wrk1:
517         begin
518 {}        reset(wrk2, 'XRF2JNK.TMP;1',, 'DPO:');
519 {}        rewrite(wrk1, 'XRF1JNK.TMP;1',, 'DPO:');
520           PrintTables(wrk2, wrk1)
521         end
522     end {  case  };
523     if OutputSection <> idents
524     then
525       begin
526         ItemCnt := 0;
527         for HshTblIndx := 0 to HashTblSize do
528           HashTable[HshTblIndx].keyindx := 0;
529         for chptr := 1 to idlen do
530           StgTable[chptr] := StgTable[FreeStgPtr + chptr - 1];
531         FreeStgPtr := 1;
532       end;
533   end {  dump tables  };
534
535 procedure scan;
536
537   label
538     1 {  exit scan on eof while processing comment  };
539
540   procedure advance;
541
542     begin
543       if DisplayIsActive   then write(rawch);   get(input);
544       rawch := input^;   ch := UpperCase(input^);
545     end {  advance  };
546
547   procedure OpenLine;
548
549     begin
550       CurrentLineNumber := CurrentLineNumber + 1;
551       if DisplayIsActive
552       then
553         begin
554           if LinesOnPage >= LinesPerPage   then Pageheader;
555           write(' ', CurrentLineNumber: DigitsPerNumber, ' ');
556           LinesOnPage := LinesOnPage + 1;
557         end;
558     end {  openline  };
559
560   procedure CloseLine;
561
562     begin
563       get(input);   rawch := input^;   ch := UpperCase(input^);
564       if DisplayIsActive   then writeln;
565     end {  closeline  };
566
567   procedure enter;
568
569     var
570             hashval,
571           FstHashVal,
572         displacement: integer;
573           NewItemPtr,
574             TailPtr: ItemPtr;
575               found: Boolean;
576                   i: integer;
577
578     procedure MakeNew(var AnItemPtr: ItemPtr);
579
580       begin
581         if (ItemCnt > MaxItems) or (FreeStgPtr > StgTblLimit)
582         then DumpTables;
583         if FreeItemPtr = nil   then new(AnItemPtr)
584         else
585           begin
586             AnItemPtr := FreeItemPtr;   FreeItemPtr := AnItemPtr^.next
587           end;
588         ItemCnt := ItemCnt + 1
589       end { makenew };
590
591     begin { enter }
592       MakeNew(NewItemPtr);   hashval := 1;
593       for i := FreeStgPtr to FreeStgPtr + idlen - 1 do
594         hashval := (hashval * 17 + abs(ord(StgTable[i]) - ord('A'))) mod
595           HashTblSize;
596       FstHashVal := hashval;   found := false;   displacement := 1;
597       NewItemPtr^.LineNumber := CurrentLineNumber;
598       repeat
599         if HashTable[hashval].keyindx = 0
600         then
601           begin {  new entry  }
602             found := true;   HashTable[hashval].keyindx := FreeStgPtr;
603             HashTable[hashval].keylen := idlen;
604             FreeStgPtr := FreeStgPtr + idlen;
605             HashTable[hashval].lastptr := NewItemPtr;
606             NewItemPtr^.next := NewItemPtr;
607           end
608         else
609           if EqlStg(FreeStgPtr, idlen, HashTable[hashval].keyindx,
610             HashTable[hashval].keylen)
611           then
612             begin {  found  }
613               found := true;   TailPtr := HashTable[hashval].lastptr;
614               NewItemPtr^.next := TailPtr^.next;
615               TailPtr^.next := NewItemPtr;
616               HashTable[hashval].lastptr := NewItemPtr;
617             end
618           else
619             begin {  collision  }
620               hashval := (hashval + displacement) mod HashTblSize;
621               displacement := displacement + 2;
622               if displacement > 2 * HashTblSize then
623                 begin
624                   DumpTables;   hashval := FstHashVal;
625                   displacement := 1;
626                 {  start over  }
627                 end
628             end
629       until found
630     end {  enter  };
631
632   begin {  scan  }
633     if eof(input)
634     then begin writeln(tty, ' Empty input file.');   goto 1; end;
635     rawch := input^;   ch := UpperCase(input^);
636     while not eof(input) do
637       begin
638         OpenLine;
639         while not eoln(input) do
640           begin
641             if ch = ' '   then advance
642             else
643               if ChrCatagory[ch] in [ucLetter, lcLetter]
644               then
645                 begin
646                   charindx := 0;   idlen := 0;   identifier := empty;
647                   fstchar := ch;
648                   repeat
649                     if charindx < keylength then
650                       begin
651                         charindx := charindx + 1;
652                         identifier[charindx] := ch;
653                       end;
654                     if AllCapitals
655                     then StgTable[FreeStgPtr + idlen] := ch
656                     else StgTable[FreeStgPtr + idlen] := rawch;
657                     idlen := idlen + 1;   advance
658                   until not (ChrCatagory[ch] in [ucLetter, digit]);
659                   lstchar := identifier[charindx];
660                   if idlen > charindx   then enter
```

```
661                 else
662                   if identifier <> reservedCcharindx +
663                      ReservRepresentedByCfstchar] +
664                      ReservRepresentedByClstchar]] {  perfect hash  }
665                   then
666                     if DoPredefined  then enter
667                     else
668                       if identifier <> predefinedCcharindx +
669                          PredefRepresentedByCfstchar] +
670                          PredefRepresentedByClstchar]]
671                       then enter;
672               end
673             else
674               if ChrCatagoryCch] = digit
675               then
676                 repeat advance;   if ch = '.'  then advance
677                 until (ChrCatagoryCch] <> digit) and (ch <> 'E')
678 {}                 and (ch <> 'B') and (ch <> 'e')
679               else
680                 if ch = quote
681                 then
682                   begin {  string  }
683                     repeat advance
684                     until (ch = quote) or eoln(input);
685                     if not eoln(input)  then advance
686                   end
687                 else
688                   if ch = lCurleyBra
689                   then
690                     begin {  comment  }
691                       advance;
692                       while ch <> rCurleyBra do
693                         begin
694                           advance;
695                           while eoln(input) do
696                             begin
697                               CloseLine;
698                               if eof(input)  then goto 1
699                               else OpenLine
700                             end
701                         end;
702                       advance
703                     end
704                   else
705                     if ch = '('
706                     then
707                       begin
708                         advance;
709                         if ch = '*'
710                         then
711                           begin {  comment  }
712                             advance;
713                             repeat
714                               while ch <> '*' do
715                                 begin
716                                   if eoln(input)
717                                   then
718                                     repeat
719                                       CloseLine;
720                                       if eof(input)  then goto 1
721                                       else OpenLine
722                                     until not eoln(input)
723                                   else advance
724                                 end;
725                               advance
726                             until ch = ')';
727                             advance
728                           end
729                       end
730                     else advance
731         end;
732       CloseLine
733     end;
734 1: {  terminate scan on eof while processing comment  }
735  end {  scan  };
736
737  {$Y+  new segment  }
738
739 procedure initialize;
740
741   procedure InitLetDig;
742
743     const
744       MinCharOrd = 0;
745       {  ordinal of minimum character  }
746       DefaultMaxCharOrd = 64;
747       {  BCD = 64 & ASCII = 127 & EBCDIC = 255  }
748
749     var
750                   i,
751           MaxCharOrd: integer;
752                   ch: char;
753
754     procedure InitChrVal(StartChar, endchar: char; aval: integer);
755
756       var
757                 lcChar,
758                 ucChar: char;
759
760       begin
761         for lcChar := StartChar to endchar do
762           begin
763             ChrCatagoryClcChar] := lcLetter;
764             ChrSortOrdClcChar] := aval;   ucChar := UpperCase(lcChar);
765             ChrCatagoryCucChar] := ucLetter;
766             ChrSortOrdCucChar] := aval - 1;   aval := aval + 2;
767           end
768       end { initchrval };
769
770     begin { initletdig }
771       if ord('A') = 193  then {  EBCDIC  } MaxCharOrd := 255
772       else
773         if ord('A') = 65  then {  ASCII  } MaxCharOrd := 127
774         else MaxCharOrd := DefaultMaxCharOrd;
775       for i := MinCharOrd to MaxCharOrd do
776         begin ChrCatagoryCchr(i)] := other;   ChrSortOrdCchr(i)] := 0;
777         end;
778       for ch := '0' to '9' do
779         begin
780           ChrCatagoryCch] := digit;
781           ChrSortOrdCch] := 100 + ord(ch) - ord('0');
782         end;
783        {  Should work for all Pascal compatible character sets  }
784        {  which are contiguous and for EBCDIC as well.          }
785       InitChrVal('a', 'i', 2);   InitChrVal('j', 'r', 20);
786       InitChrVal('s', 'z', 38);
787     end { initletdig };
788
789   procedure InitPerfect;
790
791     procedure InitReserved;
792
793       var
794                   ch: char;
795
796       begin {    RJC's perfect hash function  }
797       {    for Pascal's reserved words and predefined identifiers    }
798       {    table index = identifier length +       }
799       {          reservrepresentedby[identifier's first character] +  }
800       {          reservrepresentedby[identifier's last  character]    }
801         for ch := '0' to '9' do ReservRepresentedByCch] := 0;
802         ReservRepresentedBy['A'] := 11;
803         ReservRepresentedBy['B'] := 15;  ReservRepresentedBy['C'] := 1;
804         ReservRepresentedBy['D'] := 0;   ReservRepresentedBy['E'] := 0;
805         ReservRepresentedBy['F'] := 15;  ReservRepresentedBy['G'] := 3;
806         ReservRepresentedBy['H'] := 15;
807         ReservRepresentedBy['I'] := 13;  ReservRepresentedBy['J'] := 0;
808         ReservRepresentedBy['K'] := 0;   ReservRepresentedBy['L'] := 15;
809         ReservRepresentedBy['M'] := 15;
810         ReservRepresentedBy['N'] := 13;  ReservRepresentedBy['O'] := 0;
811         ReservRepresentedBy['P'] := 15;  ReservRepresentedBy['Q'] := 0;
812         ReservRepresentedBy['R'] := 14;  ReservRepresentedBy['S'] := 6;
813         ReservRepresentedBy['T'] := 6;   ReservRepresentedBy['U'] := 14;
814         ReservRepresentedBy['V'] := 10;  ReservRepresentedBy['W'] := 6;
815         ReservRepresentedBy['X'] := 0;   ReservRepresentedBy['Y'] := 13;
816         ReservRepresentedBy['Z'] := 0;   reserved[1] := empty;
817         reserved[38] := empty;           reserved[39] := empty;
818         reserved[40] := empty;
819 {}      ch := 'A' {  prevent optimizing 'and' to empty - compile bug  };
820         reserved[14] := 'AND      ';   reserved[29] := 'ARRAY    ';
821         reserved[33] := 'BEGIN    ';   reserved[ 5] := 'CASE     ';
822         reserved[12] := 'CONST    ';   reserved[13] := 'DIV      ';
823         reserved[ 2] := 'DO       ';   reserved[ 6] := 'DOWNTO   ';
824         reserved[ 4] := 'ELSE     ';   reserved[ 3] := 'END      ';
825         reserved[19] := 'FILE     ';   reserved[32] := 'FOR      ';
826         reserved[36] := 'FUNCTION ';
827         reserved[ 7] := 'GOTO     ' {  to xref gotos set to empty  };
828         reserved[30] := 'IF       ';   reserved[28] := 'IN       ';
829         reserved[35] := 'LABEL    ';   reserved[18] := 'MOD      ';
830         reserved[31] := 'NIL      ';   reserved[22] := 'NOT      ';
831         reserved[17] := 'OF       ';
832         {  if otherwise becomes reserved then flush left the next.  }
833         reserved[ 9] := ' OTHERWISE'
834            {  anticipating the revised standard  };
835         reserved[16] := 'OR       ';   reserved[21] := 'PACKED   ';
836         reserved[24] := 'PROCEDURE ';   reserved[37] := 'PROGRAM  ';
837         reserved[20] := 'RECORD   ';   reserved[26] := 'REPEAT   ';
838         reserved[15] := 'SET      ';   reserved[23] := 'THEN     ';
839         reserved[ 8] := 'TO       ';   reserved[10] := 'TYPE     ';
840         reserved[34] := 'UNTIL    ';   reserved[27] := 'VAR      ';
841         reserved[11] := 'WHILE    ';   reserved[25] := 'WITH     ';
842       end {  initreserved  };
843
844     procedure InitStates;
845
846       begin
847         NextStateCinout, true] := inout;
848         NextStateCinout, false] := inwrk1;
849         NextStateCinwrk1, true] := wrk1out;
850         NextStateCinwrk1, false] := wrk1wrk2;
851         NextStateCwrk1out, true] := wrk1out;
852         NextStateCwrk1out, false] := wrk1out;
853         NextStateCwrk1wrk2, true] := wrk2out;
854         NextStateCwrk1wrk2, false] := wrk2wrk1;
855         NextStateCwrk2out, true] := wrk2out;
856         NextStateCwrk2out, false] := wrk2out;
857         NextStateCwrk2wrk1, true] := wrk1out;
858         NextStateCwrk2wrk1, false] := wrk1wrk2;   state := inout;
859       end { initstates };
860
861   {$Y+  new segment  }
862
863     procedure InitPredefined;
864
865       var
866                   ch: char;
867
868       begin
869         for ch := '0' to '9' do PredefRepresentedByCch] := 0;
870         PredefRepresentedBy['A'] := 15;  PredefRepresentedBy['B'] := 9;
871         PredefRepresentedBy['C'] := 11;
872         PredefRepresentedBy['D'] := 19;  PredefRepresentedBy['E'] := 5;
873         PredefRepresentedBy['F'] := 3;   PredefRepresentedBy['G'] := 0;
874         PredefRepresentedBy['H'] := 0;   PredefRepresentedBy['I'] := 3;
875         PredefRepresentedBy['J'] := 0;   PredefRepresentedBy['K'] := 16;
876         PredefRepresentedBy['L'] := 13;  PredefRepresentedBy['M'] := 1;
877         PredefRepresentedBy['N'] := 19;  PredefRepresentedBy['O'] := 0;
878         PredefRepresentedBy['P'] := 18;  PredefRepresentedBy['Q'] := 0;
879         PredefRepresentedBy['R'] := 0;   PredefRepresentedBy['S'] := 15;
880         PredefRepresentedBy['T'] := 0;   PredefRepresentedBy['U'] := 17;
```

```
881         PredefRepresentedBy['V'] := 0;   PredefRepresentedBy['W'] := 10;
882         PredefRepresentedBy['X'] := 0;   PredefRepresentedBy['Y'] := 0;
883         PredefRepresentedBy['Z'] := 0;   predefined[1] := empty;
884         predefined[ 2] := empty;   predefined[ 3] := empty;
885         predefined[42] := empty;   predefined[43] := empty;
886         predefined[44] := empty;   predefined[45] := empty;
887         predefined[46] := empty;   predefined[47] := empty;
888         predefined[48] := empty;
889 {}      ch := 'A' {  prevent optimizing 'abs' to empty - compile bug  };
890         predefined[33] := 'ABS      ';
891         predefined[40] := 'ARCTAN   ';
892         predefined[35] := 'BOOLEAN  ';
893         predefined[15] := 'CHAR     ';
894         predefined[14] := 'CHR      ';
895         predefined[29] := 'COS      ';
896         predefined[31] := 'DISPOSE  ';
897         predefined[11] := 'EOF      ';
898         predefined[28] := 'EOLN     ';
899         predefined[26] := 'EXP      ';
900         predefined[13] := 'FALSE    ';  predefined[3] := 'GET      ';
901         predefined[ 8] := 'INPUT    ';  predefined[10] := 'INTEGER  ';
902         predefined[34] := 'LN       ';  predefined[7] := 'MAXINT   ';
903         predefined[32] := 'NEW      ';
904         predefined[22] := 'ORD      ';  predefined[6] := 'OUTPUT   ';
905         predefined[38] := 'PACK     ';
906         predefined[27] := 'PAGE     ';
907         predefined[41] := 'PRED     ';
908         predefined[21] := 'PUT      ';
909         predefined[23] := 'READ     ';
910         predefined[25] := 'READLN   ';
911         predefined[17] := 'REAL     ';  predefined[5] := 'RESET    ';
912         predefined[12] := 'REWRITE  ';
913         predefined[24] := 'ROUND    ';
914         predefined[37] := 'SIN      ';
915         predefined[18] := 'SQR      ';
916         predefined[19] := 'SQRT     ';
917         predefined[30] := 'SUCC     ';  predefined[4] := 'TEXT     ';
918         predefined[ 9] := 'TRUE     ';  predefined[16] := 'TRUNC    ';
919         predefined[39] := 'UNPACK   ';
920         predefined[20] := 'WRITE    ';
921         predefined[36] := 'WRITELN  ';
922       end {  initpredefined  };
923
924   {$Y+ new segment  }
925
926     begin {  initperfect  }
927       InitReserved;   InitStates;   InitPredefined;
928     end {  initperfect  };
929
930 {}  procedure ConnectFiles;
931
932 {}    const
933 {}      FSpecLeng = 32;
934
935 {}    type
936 {}              fspecs = array [1 .. FSpecLeng] of char;
937 {}           FileSpecs = array [1 .. 2] of fspecs;
938 {}           extension = packed array [1 .. 4] of char;
939 {}           FileNames = array [1 .. 32] of char;
940 {}                devs = array [1 .. 5] of char;
941 {}                dirs = array [1 .. 9] of char;
942 {}                nams = array [1 .. 18] of char;
943
944 {}    var
945 {}                fspec: FileSpecs;
946 {}                 flen: 0 .. FSpecLeng;
947 {}               cmlptr: 1 .. 80;
948 {}                CmlCh,
949 {}                CmdCh: char;
950 {}              DotFound: Boolean;
951 {}                 pos: integer;
952
953
954
955 {} procedure SplitFileSpecification (fspec: FileNames;
956 {}                   var dev: devs; var  dir: dirs; var  nam: nams);
957 {}    label   2;
958
959 {}    var
960 {}      i: 1 .. 33;   j: 1 .. 19;
961
962 {}    begin
963 {}      dev := '     '; dir := '         ';
964 {}      nam := '                  ';
965
966 {}      for i := 1 to 32 do
967 {}          if fspec[i] >= 'a' then
968 {}            fspec[i] := CHR (ord(fspec[i]) - 40B);
969
970 {}      i := 1;
971 {}      while not (fspec[i] in [':', '[', '.', ';']) and (i < 32) do
972 {}      begin
973 {}          i := i + 1;
974 {}      end;
975
976 {}      if fspec[i] = ':' then   {  contains a device name  }
977 {}      begin
978 {}          for j := 1 to i do
979 {}            if j <= 5 then dev[j] := fspec[j];
980 {}          i := i + 1;
981 {}      end
982 {}      else i := 1;
983
984 {}      if fspec[i] = '[' then   {  contains a directory part  }
985 {}      begin
986 {}          j := 1;
987 {}          repeat
988 {}            dir[j] := fspec[i];
989 {}            i := i + 1;   j := j + 1;
990 {}          until (i > 32) or (j > 9) or (dir[j-1] = ']');
991 {}      end;
992
993 {}      if i <= 32 then  {  contains file name part  }
994 {}      begin
995 {}          j := 1;
996 {}          while ord(fspec[i]) > ord(' ') do
997 {}          begin
998 {}            nam[j] := fspec[i];
999 {}            i := i + 1;   j := j + 1;
1000 {}           if (i > 32) or (j > 18) then  goto 2;
1001 {}         end;
1002 {}2:
1003 {}     end;
1004 {}   end;
1005
1006
1007
1008 {} procedure reeset (var f: text;  var fspec: FileNames);
1009
1010 {}   var
1011 {}     dev:devs;   dir: dirs;   nam: nams;
1012
1013 {}   begin
1014 {}     SplitFileSpecification (fspec, dev, dir, nam);
1015 {}     reset (f, nam, dir, dev);
1016 {}   end;
1017
1018
1019
1020
1021 {} procedure reewrite (var f: text;  var fspec: FileNames);
1022
1023 {}   var
1024 {}     dev:devs;   dir: dirs;   nam: nams;
1025
1026 {}   begin
1027 {}     SplitFileSpecification (fspec, dev, dir, nam);
1028 {}     rewrite (f, nam, dir, dev);
1029 {}   end;
1030
1031 {}   procedure GCML(var line: LineBuffer; var len: integer);
1032 {}     extern  {  return command line in upper case  };
1033
1034 {}  procedure quit;
1035
1036 {}   begin
1037 {}     writeln(tty, ' Errors in Command Line');
1038 {}     for cmlptr := 1 to cmllen do write(tty, cmllline[cmlptr]);
1039 {}     writeln(tty);   writeln(tty);
1040 {}     writeln(tty, ' <output file>=<input file> [<optons>]');
1041 {}     writeln(tty, ' <options> ::=');
1042 {}     writeln(tty, '   C-    capitalize identifiers,');
1043 {}     writeln(tty, '   D+    display program,');
1044 {}     writeln(tty, '   P-    cross ref predefined ids.,');
1045 {}     writeln(tty, '   T-    terminal output (ids. only),');
1046 {}     writeln(tty, '   W=132 width of output.');   writeln(tty);
1047 {}     writeln(tty);   writeln(tty, ' HALT');   halt
1048 {}   end { quit };
1049
1050 {}  procedure NextClmCh;
1051
1052 {}   begin
1053 {}     if cmlptr >= cmllen   then quit;   cmlptr := cmlptr + 1;
1054 {}     CmlCh := cmllline[cmlptr]
1055 {}   end {  nextcmlch  };
1056
1057 {}  procedure getfspec(InputOutput: integer; DefaultExtension: extension
1058 {}     );
1059
1060 {}   procedure getnext;
1061
1062 {}     begin
1063 {}       if flen >= FSpecLeng   then quit;
1064 {}       fspec[InputOutput] [flen] := CmlCh;   flen := flen + 1;
1065 {}       NextClmCh;
1066 {}     end {  getnext  };
1067
1068 {}   begin {  getfspec  }
1069 {}     fspec[InputOutput] := '                              ';
1070 {}     flen := 1;   DotFound := false;
1071 {}     while CmlCh in ['A' .. 'Z', '0' .. '9', ':', '[', ']', '.', ',',
1072 {}       ';'] do
1073 {}       if CmlCh = '['   then repeat getnext;   until CmlCh = ']'
1074 {}       else
1075 {}         begin
1076 {}           if not DotFound   then DotFound := CmlCh = '.';   getnext;
1077 {}         end;
1078 {}     if (flen > 1) and (not DotFound) then
1079 {}       for pos := 1 to 4 do
1080 {}         begin
1081 {}           fspec[InputOutput] [flen] := DefaultExtension[pos];
1082 {}           flen := flen + 1;
1083 {}         end;
1084 {}   end {  getfspec  };
1085
1086 {}  begin {  connectfiles  }
1087 {}    GCML(cmlline, cmllen);   CmlCh := cmllline[1];   cmlptr := 1;
1088 {}    cmllen := cmllen + 1;   cmllline[cmllen] := ' ';
1089 {}    while CmlCh <> ' ' do NextClmCh;   while CmlCh = ' ' do NextClmCh;
1090 {}    getfspec(1, '.LST');
1091 {}    if flen = 1
1092 {}    then begin writeln(tty, ' No Output File Specified');   quit; end;
1093 {}    NextClmCh;   while CmlCh = ' ' do NextClmCh;
1094 {}    getfspec(2, '.PAS');
1095 {}    if flen = 1
1096 {}    then begin writeln(tty, ' No Input File Specified');   quit; end;
1097 {}    reeset(input, fspec[2]);   reewrite(output, fspec[1]);
1098 {}    while ((cmlptr < cmllen) and (CmlCh <> '[')) do NextClmCh;
1099 {}    if CmlCh = '['
1100 {}    then
```

```
{}      repeat
{}        NextClmCh;
{}        while (CmlCh = ' ') or (CmlCh = ',') do NextClmCh;
{}        if CmlCh in ['C', 'D', 'P', 'T', 'W']
{}        then
{}          begin
{}            CmdCh := CmlCh;   NextClmCh;
{}            case CmdCh of
{}              'C': AllCapitals := CmlCh = '+';
{}              'D': DisplayIsActive := CmlCh = '+';
{}              'P': DoPredefined := CmlCh = '+';
{}              'T':
{}                begin
{}                  terminal := CmlCh = '+';
{}                  if terminal
{}                  then LineLength := DefaultTerminalWidth;
{}                  DisplayIsActive := not terminal;
{}                end;
{}              'W':
{}                begin
{}                  if (CmlCh = ':') or (CmlCh = '=')   then NextClmCh;
{}                  LineLength := 0;
{}                  while CmlCh in ['0' .. '9'] do
{}                    begin
{}                      LineLength := LineLength * 10 + ord(CmlCh) - ord
{}                          ('0');
{}                      NextClmCh;
{}                    end;
{}                  if LineLength < (DefaultTerminalWidth - 8)
{}                  then LineLength := DefaultLpWidth;
{}                end
{}            end;
{}        until CmlCh = ']';
{}      end {  connectfiles  };
{}
{}    begin {  initialize  }
{}      CurrentLineNumber := 0;   PageNumber := 0;
{}      LinesOnPage := LinesPerPage;   AllCapitals := false;
{}      DisplayIsActive := true;   DoPredefined := false;   FreeStgPtr := 1;
{}      FreeItemPtr := nil;
{}      for ItemCnt := 1 to 80 do cmlline[ItemCnt] := ' ';   cmllen := 0;
{}      ItemCnt := 0;   terminal := false;   empty := '          ';
{}      for HshTblIndx := 0 to HashTblSize - 1 do
{}        HashTable[HshTblIndx].keyindx := 0;
{}      InitLetDig;   InitPerfect;   LineLength := DefaultLpWidth;
{}      today := empty;   now := empty;
{}      ConnectFiles;   date(today);   time(now);
{}      wrk2active := false;
{}    end {  initialize  };
{}
{}    {$Y+  new segment  }
{}
{}  begin {  xref  }
{}    writeln(tty, '- CrossRef (80.2.1)');   initialize;
{}    OutputSection := listing;   scan;   OutputSection := idents;
{}    DumpTables;   writeln(tty, '- End CrossRef');   writeln(tty, ' ');
{}  end {  xref  }.
```

```
1   {* Purpose:
2        Library routines for string manipulation.
3
4     * Author:
5        Barry Smith
6        Oregon Software
7        2340 SW Canyon Road
8        Portland Oregon 97201
9
10    * Method:
11       Uses fixed length arrays of characters.
12
13    * Description of Routines:
14       Len         -- Function.  Returns string length.
15       Clear       -- Blank fills a string.
16       Concatenate -- Appends one string to another.
17       Search      -- Function.  Returns substring position.
18       Readstring  -- Read a string from a file.
19       Writestring -- Write a string to a file.
20       Substring   -- Extract a substring from a string.
21       Delete      -- Remove part of a string.
22       Insert      -- Insert a string into a string.
23
24     In several routines error processing is left for the
25    user to provide.
26
27    * Computer System:
28       DEC PDP 11, OMSI Pascal version 1.
29
30    *}
31
32  const
33    stringmax = 100;
34
35  type
36    string = record
37               len: 0 .. stringmax;
38               ch: packed array [1 .. stringmax] of char
39             end;
40
41  function len(s: string): integer;
42
43    begin len := s.len end { len };
44
45  procedure clear(var s: string);
46
47    var
48      i: integer;
49
50    begin s.len := 0;   for i := 1 to stringmax do s.ch[i] := ' '
51    end { clear };
52
53  procedure concatenate(var s: string; t: string);
54
55    var
56      i, j: integer;
57
58    begin
59      if s.len + t.len > stringmax
60      then j := stringmax - s.len { overflow }
61      else j := t.len;
62      for i := 1 to j do s.ch[s.len + i] := t.ch[i];   s.len := s.len + j;
63    end { concatenate };
64
65  function search(s, t: string; start: integer): integer;
66
67    var
68      i, j: 0 .. stringmax;
69      uneq: boolean;
70
71    begin
72      if start < 1   then start := 1;
73      if (start + t.len > s.len + 1) or (t.len = 0)   then search := 0
74      else
75        begin
76          i := start - 1;
77          repeat
78            i := i + 1;   j := 0;
79            repeat j := j + 1;   uneq := t.ch[j] <> s.ch[i + j - 1];
80            until uneq or (j = t.
81            len);
82          until (not uneq) or (i = s.
83          len - t.len + 1);
84          if uneq   then search := 0   else search := i;
85        end;
86    end { search };
87
88  procedure readstring(var f: text; var s: string);
89
90    begin
91      clear(s);
92      with s do
93        while (not eoln(f)) and (len < stringmax) do
94          begin len := len + 1;   read(f, ch[len]); end;
95      readln(f);
96    end { readstring };
97
98  procedure writestring(var f: text; s: string);
99
100   var
101     i: integer;
102
103   begin for i := 1 to s.len do write(f, s.ch[i]) end { writestring };
104
105 procedure substring(var t: string; s: string; start, span: integer);
106
107   var
108     i: integer;
109
110   begin
111     if span < 0
112     then begin span := - span;   start := start - span end;
113     if start < 1
114     then begin span := span + start - 1;   start := 1 end;
115     if start + span > s.len + 1   then span := s.len - start + 1;
116     if span <= 0   then clear(t)
117     else
118       begin
119         for i := 1 to span do t.ch[i] := s.ch[start + i - 1];
120         for i := span + 1 to stringmax do t.ch[i] := ' ';
121         t.len := span;
122       end;
123   end { substring };
124
125 procedure delete(var s: string; start, span: integer);
126
127   var
128     i, limit: integer;
129
130   begin
131     if span < 0
132     then begin span := - span;   start := start - span end;
133     limit := start + span;   if start < 1   then start := 1;
134     if limit > s.len + 1   then limit := s.len + 1;
135     span := limit - start;
136     if span > 0
137     then
138       begin
139         for i := 0 to s.len - limit do
140           s.ch[start + i] := s.ch[limit + i];
141         for i := s.len - span + 1 to s.len do s.ch[i] := ' ';
142         s.len := s.len - span;
143       end;
144   end { delete };
145
146 procedure insert(var s: string; t: string; p: integer);
147
148   var
149     i, j: integer;
150
151   begin
152     if t.len > 0
153     then
154       if (p > 0) and (p <= s.len + 1)
155       then
156         begin
157           if s.len + t.len <= stringmax   then s.len := s.len + t.len
158           else s.len := stringmax { overflow };
159           for i := s.len downto p + t.len do s.ch[i] := s.ch[i - t.len];
160           if s.len < p + t.len   then j := s.len
161           else j := p + t.len - 1;
162           for i := p to j do s.ch[i] := t.ch[i - p + 1];
163         end
164       else { non-contiguous string }
165   end { insert };
166
```

```
{* Purpose:
     Program computes Hankel functions of the first and second
     kinds for an integrel order and complex argument.

 * Author:
     Q.M. Tran, School of Electrical Engineering, University of New
     South Wales.

 * Method:
     Hankel functions of a required order are calculated from
     corresponding Bessel functions of the first and second kinds.
     A backward recursive scheme is used in computing Bessel function
     of the first kind for a number of orders.
     These are then summed to give the two orders 0 and 1 of
     Bessel function of the second kind, which in turn serve as
     starting point for finding a higher-order Bessel function of
     the second kind.

 * Description of parameters:
     p   - integral order, where -max <= p <= max and max = 500.
     z   - complex argument.
     fn1 - Hankel function of the first kind.
     fn2 - Hankel function of the second kind.

 * Input:
     Program reads in an integer (p) and two real numbers (real and
     imaginary parts of z).

 * Output:
     Arguments and values of the Hankel functions of the first
     and second kinds are returned.
     Warning message is given if any parameter exceeds specified
     limits or is outside range.

 * Limitations:
     - 500 <= p <= 500 ,
     1.0E-5 <= modulus of z <= 377.0 ,
     Imaginary part of z <= 50.0 ,
         p must not be much greater than the modulus of z, otherwise
     exponent error in the computer (PDP 11/70) will occur.

 * Computer system:
     Program was run under UNIX Pascal (Berkeley - Version 1.2,
     May 1979) on DEC PDP 11/70.

 * Accuracy:
     Computed results were checked against published values over the
     following ranges:
         - 100 <= p <= 100 and
             real argument z = 0.1 - 100.0 ,
         - 1 <= p <= 1 and
             complex argument z = (0.01,5 deg.) - (10.0,90 deg.)

     They were found to be accurate to at least 10 significant digits. }


program hankel(input, output);

label
  1 { Exit to terminate program };

const
  lim = 501;
  max = 500;
  tpi = 0.6366197723675813 { 2.0 by pi };
  euler = 0.5772156649015329;

type
  complex = record
              re, im: real
            end;

var
  i, k, n, m, l, p: integer;
  z, u, v, w, yo, y1, y2: complex;
  fn1, fn2, sum, esum, osum, norm, zero: complex;
  f: array [0 .. lim] of complex;


procedure stop;

  begin
    goto 1 { halt }
  end { stop };


procedure cread(var z: complex);

  begin
    read(z.re, z.im)
  end { cread };


procedure cwrite(var z: complex);

  begin
    writeln('(', z.re, ',', z.im, ')')
  end { cwrite };


function mag(var z: complex): real;
{ Computes the modulus of a complex number }

  begin
    mag := sqrt(sqr(z.re) + sqr(z.im))
  end { mag };


procedure add(u, v: complex; var w: complex);
  begin
    w.re := u.re + v.re;   w.im := u.im + v.im
  end { add };


procedure sub(u, v: complex; var w: complex);

  begin
    w.re := u.re - v.re;   w.im := u.im - v.im
  end { sub };


procedure mult(a: real; z: complex; var w: complex);
{ Multiplies a real with a complex }

  begin
    w.re := a * z.re;   w.im := a * z.im
  end { mult };


procedure product(u, v: complex; var w: complex);

  begin
    w.re := (u.re * v.re) - (u.im * v.im);
    w.im := (u.re * v.im) + (u.im * v.re)
  end { product };


procedure quotient(u, v: complex; var w: complex);

  var
    vr, vi, a, b, x1, x2, y1, y2, root: real;

  begin
    vr := abs(v.re);   vi := abs(v.im);
    root := sqrt(2.0) * sqrt(vr) * sqrt(vi);   a := vr + vi + root;
    b := vr + vi - root;
    if (a = 0.0) or (b = 0.0) then
      begin
        writeln('W: dividing by 0 in procedure quotient');
        stop { Exit to terminate program };
      end;
    x1 := u.re / a;   x2 := v.re / b;   y1 := u.im / a;
    y2 := v.im / b;   w.re := x1 * x2 + y1 * y2;
    w.im := x2 * y1 - x1 * y2
  end { quotient };


procedure ccos(z: complex; var c: complex);
{ Cosine of a complex }

  var
    ep, em, p, m: real;

  begin
    ep := exp(z.im);   em := 1.0 / ep;   p := ep + em;   m := em - ep;
    c.re := 0.5 * p * cos(z.re);   c.im := 0.5 * m * sin(z.re)
  end { ccos };


procedure polar(u: complex; var v: complex);
{ Writing a complex into polar form }

  const
    pi = 3.1415926535897932;

  begin
    if (u.re = 0.0) and (u.im = 0.0) then
      begin
        writeln('W: conversion of 0 in procedure polar');
        stop { Exit to terminate program };
      end;
    if (u.re = 0.0) and (u.im <> 0.0) then
      begin
        v.re := mag(u);   v.im := pi / 2.0
      end
    else
      begin
        v.re := mag(u);   v.im := arctan(u.im / u.re)
      end
  end { polar };


procedure cln(z: complex; var c: complex);
{ Natural logarithm of a complex }

  var
    p: complex;

  begin
    polar(z, p);   c.re := ln(p.re);   c.im := p.im
  end { cln };


function order(z: complex): integer;
{ Gives a starting and even order for recursive computation }

  var
    a: real;
    m: integer;

  begin
    a := mag(z);
    if a < 0.1   then m := 10
    else
      begin if a < 2.0   then m := 28   else m := round(1.2 * a + 48.0)
      end;
    order := m;   if odd(m)   then order := m + 1
  end { order };
```

```
221
222 procedure sign(u: complex; var v: complex);
223 { Changes the sign of a complex }
224
225   begin
226     v.re := - u.re;   v.im := - u.im
227   end { sign };
228
229
230 procedure check(z: complex);
231 { Checks to see if the function argument is outside range }
232
233   var
234     a, b: real;
235
236   begin
237     a := abs(z.re);   b := abs(z.im);
238     if ((a < 1.0E - 5) and (b < 1.0E - 5)) or
239        ((b <> 0.0) and (b < 1.0E - 5)) then
240       begin
241         write('W: small argument which causes exponent error = ');
242         cwrite(z);   stop { Exit to terminate program };
243       end;
244     if b > 50.0 then
245       begin
246         write('W: argument with imaginary part outside range = ');
247         cwrite(z);   stop { Exit to terminate program };
248       end
249   end { check };
250
251
252 procedure hankel12(u, v: complex; var w1, w2: complex);
253 { Combines Bessel functions of the first & second kinds to give Hankel
254   functions }
255
256   begin
257     w1.re := u.re - v.im;   w1.im := u.im + v.re;
258     w2.re := u.re + v.im;   w2.im := u.im - v.re
259   end { hankel12 };
260
261
262 begin { Hankel }
263   read(p);   n := abs(p);
264   if n >= lim then
265     begin
266       writeln('W: required order ', p: 6, ' is outside the range (', -
267           max: 4, ',', max: 4, ')');
268       stop { Exit to terminate program };
269     end;
270   cread(z);
271   check(z) { If z is outside range, exit to terminate program };
272   m := order(z);
273   if m >= lim then
274     begin
275       writeln('W: starting order ', m: 6,
276           ' exceeds the specified maximum', max: 4);
277       stop { Exit to terminate program };
278     end;
279   zero.re := 0.0;   zero.im := 0.0;   sum := zero;   esum := zero;
280   osum := zero;   f[m + 1] := zero;   f[m].re := 1.0e - 30;
281   f[m].im := 0.0;
282   for i := m downto 1 do
283     begin
284       quotient(f[i], z, w);   mult(2.0 * i, w, w);
285       sub(w, f[i + 1], f[i - 1])
286     end;
287   k := m div 2;
288   if abs(z.re) > 10.0 * abs(z.im)
289   then
290     begin
291       for i := 1 to k do add(sum, f[2 * i], sum);   mult(2.0, sum, sum);
292       add(sum, f[0], norm)
293     end
294   else
295     begin
296       for i := 1 to k do
297         begin
298           if odd(i)   then add(osum, f[2 * i], osum)
299           else add(esum, f[2 * i], esum)
300         end;
301       sub(esum, osum, sum);   mult(2.0, sum, sum);
302       add(sum, f[0], sum);   ccos(z, u);   quotient(sum, u, norm)
303     end;
304   for i := 0 to m do
305     quotient(f[i], norm, f[i]) { Bessel functions of 1st kind };
306   esum := zero;   osum := zero;   l := 1;
307   if n = 0
308   then
309     begin { Ho }
310       for i := 1 to k do
311         begin
312           l := - 1;   mult(l / i, f[2 * i], u);   add(esum, u, esum)
313         end;
314       mult(2.0, esum, esum);   mult(0.5, z, u);   cln(u, u);
315       u.re := u.re + euler;   product(u, f[0], u);   sub(u, esum, u);
316       mult(tpi, u, yo) { Yo };   hankel12(f[0], yo, fn1, fn2);
317       writeln;   writeln;   write('     Function argument = ');
318       cwrite(z);   writeln;
319       write('     Hankel function of the first kind and order 0 = ');
320       cwrite(fn1);   writeln;
321       write('     Hankel function of the second kind and order 0 = ');
322       cwrite(fn2);   writeln;   writeln;
323       stop { Exit to terminate program };
324     end { Ho }
325   else
326     begin { Hn, where n <> 0 }
327       for i := 1 to k do
328         begin
329           l := - 1;   mult(l / i, f[2 * i], u);   add(esum, u, esum);
330           sub(f[2 * i - 1], f[2 * i + 1], v);   mult(l / i, v, v);
331           add(osum, v, osum);
332         end;
333       mult(2.0, esum, esum);   mult(0.5, z, u);   cln(u, u);
334       u.re := u.re + euler;   product(u, f[0], v);   sub(v, esum, v);
335       mult(tpi, v, yo) { Yo };   product(u, f[1], v);
336       quotient(f[0], z, w);   sub(v, w, w);   add(w, osum, w);
337       mult(tpi, w, y1) { Y1 };   i := 1;
338       while i < n do
339           { Forward recursion to compute Yn, where n <> 0,1 }
340         begin
341           quotient(y1, z, u);   mult(2 * i, u, u);   sub(u, yo, y2);
342           yo := y1;   y1 := y2;   i := i + 1;
343         end { Forward recursion };
344       if m < max   then  for i := m + 1 to max do f[i] := zero;
345       hankel12(f[n], y1, fn1, fn2);
346       if (p < 0) and odd(p) then
347         begin
348           sign(fn1, fn1);   sign(fn2, fn2)
349         end;
350       writeln;   writeln;   write('     Function argument = ');
351       cwrite(z);   writeln;
352       write('     Hankel function of the first kind and order ', p: 4,
353         ' = ');
354       cwrite(fn1);   writeln;
355       write('     Hankel function of the second kind and order ', p: 4,
356         ' = ');
357       cwrite(fn2);   writeln;   writeln;
358     end { Hn };
359 1:
360 end { Hankel }.
```

```
{* Purpose:
     Program computes a Bessel function of the first kind for an
     integral order and complex argument.

 * Author:
     Q.M. Tran, School of Electrical Engineering, University of
     New South Wales.

 * Method:
     Backward recurrence equation is employed to compute the function,
     starting at a higher order for which the Bessel function has a
     small value.  The starting order is calculated using an
     empirical formula.  When the function argument is mainly real,
     normalization is to unity.  If it is mainly imaginary,
     normalization involves cosine of the complex argument.

 * Description of parameters:
     p  - integral order, where -max <= p <= max and max = 500
     z  - complex argument.
     fn - Bessel function of z and order p.

 * Input:
     Program reads in an integer (p) and two real numbers
     (real and imaginary parts of z).

 * Output:
     Argument & value of the Bessel function of the first kind
     are returned.  Warning message is given if any parameter
     exceeds specified limits or is outside range.

 * Limitations:
     - 500 <= p <= 500,
     1.0e-5 <= modulus of z <= 377.0,
     Imaginary part of z <= 50.0.

 * Computer system:
     Program was run under UNIX Pascal (Berkeley - Version 1.2,
     May, 1979) on DEC PDP 11/70.

 * Accuracy:
     Computed results were checked against published values over
     the following ranges:
     - 100 <= p <= 100 and 0.1 <= modulus of z <= 100.0.
     They were found to be accurate to at least 8 decimal digits. }


program bessel1(input, output);

label
  1 { Exit to terminate program };

const
  lim = 501;
  max = 500;

type
  complex = record
              re, im: real
            end;

var
  i, k, n, m, p: integer;
  z, w, fn, sum, esum, osum, norm, zero: complex;
  f: array [0 .. lim] of complex;


procedure stop;

  begin
    goto 1 { halt }
  end { stop };


procedure cread(var z: complex);

  begin
    read(z.re, z.im)
  end { cread };


procedure cwrite(var z: complex);

  begin
    writeln('(', z.re, ',', z.im, ')')
  end { cwrite };


function mag(var z: complex): real;
{ Computes the modulus of a complex number }

  begin
    mag := sqrt(sqr(z.re) + sqr(z.im))
  end { mag };


procedure add(u, v: complex; var w: complex);

  begin
    w.re := u.re + v.re;   w.im := u.im + v.im
  end {add };


procedure sub(u, v: complex; var w: complex);

  begin
    w.re := u.re - v.re;   w.im := u.im - v.im
  end { sub };


procedure mult(a: real; z: complex; var w: complex);
{ Multiplies a real with a complex }

  begin
    w.re := a * z.re;   w.im := a * z.im
  end { mult };


procedure quotient(u, v: complex; var w: complex);

  var
    vr, vi, a, b, x1, x2, y1, y2, root: real;

  begin
    vr := abs(v.re);   vi := abs(v.im);
    root := sqrt(2.0) * sqrt(vr) * sqrt(vi);   a := vr + vi + root;
    b := vr + vi - root;
    if (a = 0.0) or (b = 0.0) then
      begin
        writeln('W: dividing by 0 in procedure quotient');
        stop { Exits to terminate program };
      end;
    x1 := u.re / a;   x2 := v.re / b;   y1 := u.im / a;
    y2 := v.im / b;   w.re := x1 * x2 + y1 * y2;
    w.im := x2 * y1 - x1 * y2
  end { quotient };


procedure ccos(z: complex; var c: complex);
{ Cosine of a complex }

  var
    ep, em, p, m: real;

  begin
    ep := exp(z.im);   em := 1.0 / ep;   p := ep + em;   m := em - ep;
    c.re := 0.5 * p * cos(z.re);   c.im := 0.5 * m * sin(z.re)
  end { ccos };


function order(z: complex): integer;
{ Gives a starting and even order for recursive computation }

  var
    a: real;
    m: integer;

  begin
    a := mag(z);
    if a < 0.1   then m := 10
    else
      begin if a < 2.0   then m := 28   else m := round(1.2 * a + 48.0)
      end;
    order := m;   if odd(m)   then order := m + 1
  end { order };


procedure sign(u: complex; var v: complex);
{ Changes the sign of a complex }

  begin
    v.re := - u.re;   v.im := - u.im
  end { sign };


procedure check(z: complex);
{ Checks to see if the function argument is outside range }

  var
    a, b: real;

  begin
    a := abs(z.re);   b := abs(z.im);
    if ((a < 1.0e - 5) and (b < 1.0e - 5)) or ((b <> 0.0) and (b < 1.0e
       - 5))
    then
      begin
        write('W: small argument which causes exponent error = ');
        cwrite(z);   stop { Exits tc terminate program };
      end;
    if b > 50.0 then
      begin
        write('W: argument with imaginary part outside range = ');
        cwrite(z);   stop { Exits to terminate program };
      end
  end { check };


begin { Bessel1 }
  read(p);   n := abs(p);
  if n >= lim then
    begin
      writeln('W: required order ', p: 6, ' is outside the range (', -
         max: 4, ',', max: 4, ')');
      stop { Exits to terminate program };
    end;
  cread(z);
  check(z) { If z is outside range, exit to terminate program };
  m := order(z);
  if m >= lim then
    begin
      writeln('W: starting order ', m: 6,
         ' exceeds the specified maximum', max: 4);
      stop { Exits to terminate program };
    end;
  if n >= m
  then
    begin
      writeln;   writeln;   write('     Function argument = ');
      cwrite(z);   writeln;
      writeln('     Bessel function of the first kind and order ', p: 4,
```

```
          ' = ( 0 , 0 )');
        writeln;   writeln;   stop { Exits to terminate program };
      end;
    zero.re := 0.0;   zero.im := 0.0;   sum := zero;   esum := zero;
    osum := zero;   f[m + 1] := zero;   f[m].re := 1.0e - 30;
    f[m].im := 0.0;
    for i := m downto 1 do
      begin
        quotient(f[i], z, w);   mult(2.0 * i, w, w);
        sub(w, f[i + 1], f[i - 1])
      end;
    k := m div 2;
    if abs(z.re) > 10.0 * abs(z.im)
    then
      begin
        for i := 1 to k do add(sum, f[2 * 1], sum);   mult(2.0, sum, sum);
        add(sum, f[0], norm)
      end
```

```
{* Purpose:
     Program computes a Bessel function of the second kind for an
     integral order and complex argument.

 * Author:
     Q.M. Tran, School of Electrical Engineering, University of New
     South Wales.

 * Method:
     Initially, a number of Bessel functions of the first kind are
     generated by backward recursion.  These are then summed to give
     the two orders 0 and 1 of the Bessel function of the second kind.
     Using forward recurrence relation based on these two orders,
     a higher order is calculated.

 * Description of parameters:
     p  - integral order, where -max <= p <= max and max = 500.
     z  - complex argument.
     fn - Bessel function of z and order p.

 * Input:
     Program reads in an integer (p) and two real numbers (real and
     imaginary parts of z).

 * Output:
     Argument & value of the Bessel function of the second kind are
     returned.  Warning message is given if any parameter exceeds
     specified limits or is outside range.

 * Limitations:
     - 500 <= p <= 500 ,
     1.0e-5 <= modulus of z <= 377.0 ,
     Imaginary part of z <= 50.0 ,
     p must not be much greater than the modulus of z, otherwise
     exponent error in the computer (PDP 11/70) will occur.

 * Computer system:
     Program was run under UNIX Pascal (Berkeley - Version 1.2,
     May 1979) on DEC PDP 11/70.

 * Accuracy:
     Computed results were checked against published values over the
     following ranges:
         - 100 <= p <= 100 and
             real argument z = 0.1 - 100.0 ,
         - 1 <= p <= 1 and
             complex argument z = (0.01,5 deg.) - (10.0,90 deg.).

     They were found to be accurate to at least 10 significant digits. }
program bessel2(input, output);

label
  1 { Exit to terminate program };

const
  lim = 501;
  max = 500;
  tpi = 0.6366197723675813 { 2.0 by pi };
  euler = 0.5772156649015329;

type
  complex = record
              re, im: real
            end;

var
  i, k, n, m, l, p: integer;
  z, u, v, w, yo, y1, y2: complex;
  fn, sum, esum, osum, norm, zero: complex;
  f: array [0 .. lim] of complex;


procedure stop;

  begin
    goto 1 { halt }
  end { stop };


procedure cread(var z: complex);

  begin
    read(z.re, z.im)
  end { cread };


procedure cwrite(var z: complex);

  begin
    writeln('(', z.re, ',', z.im, ')')
  end { cwrite };


function mag(var z: complex): real;
{ Computes the modulus of a complex number }

  begin
    mag := sqrt(sqr(z.re) + sqr(z.im))
  end { mag };
```

```
  else
    begin
      for i := 1 to k do
        begin
          if odd(i)   then add(osum, f[2 * i], osum)
          else add(esum, f[2 * i], esum)
        end;
      sub(esum, osum, sum);   mult(2.0, sum, sum);
      add(sum, f[0], sum);   ccos(z, w);   quotient(sum, w, norm)
    end;
  quotient(f[n], norm, fn);
  if (p < 0) and (odd(p))   then sign(fn, fn);   writeln;   writeln;
  write('     Function argument = ');   cwrite(z);   writeln;
  write('     Bessel function of the first kind and order ', p: 4, ' = '
    );
  cwrite(fn);   writeln;   writeln;   1:
end { Bessel1 }.
```

```


procedure add(u, v: complex; var w: complex);

  begin
    w.re := u.re + v.re;   w.im := u.im + v.im
  end { add };


procedure sub(u, v: complex; var w: complex);

  begin
    w.re := u.re - v.re;   w.im := u.im - v.im
  end { sub };


procedure mult(a: real; z: complex; var w: complex);
{ Multiplies a real with a complex }

  begin
    w.re := a * z.re;   w.im := a * z.im
  end { mult };


procedure product(u, v: complex; var w: complex);

  begin
    w.re := (u.re * v.re) - (u.im * v.im);
    w.im := (u.re * v.im) + (u.im * v.re)
  end { product };


procedure quotient(u, v: complex; var w: complex);

  var
    vr, vi, a, b, x1, x2, y1, y2, root: real;

  begin
    vr := abs(v.re);   vi := abs(v.im);
    root := sqrt(2.0) * sqrt(vr) * sqrt(vi);   a := vr + vi + root;
    b := vr + vi - root;
    if (a = 0.0) or (b = 0.0) then
      begin
        writeln('W: dividing by 0 in procedure quotient');   stop;
      { Exit to terminate program }
      end;
    x1 := u.re / a;   x2 := v.re / b;   y1 := u.im / a;
    y2 := v.im / b;   w.re := x1 * x2 + y1 * y2;
    w.im := x2 * y1 - x1 * y2
  end { quotient };


procedure ccos(z: complex; var c: complex);
{ Cosine of a complex }

  var
    ep, em, p, m: real;

  begin
    ep := exp(z.im);   em := 1.0 / ep;   p := ep + em;   m := em - ep;
    c.re := 0.5 * p * cos(z.re);   c.im := 0.5 * m * sin(z.re)
  end { ccos };


procedure polar(u: complex; var v: complex);
{ Writing a complex into polar form }

  const
    pi = 3.1415926535897932;

  begin
    if (u.re = 0.0) and (u.im = 0.0) then
      begin
        writeln('W: conversion of 0 in procedure polar');   stop;
      { Exit to terminate program }
      end;
    if (u.re = 0.0) and (u.im <> 0.0) then
      begin
        v.re := mag(u);   v.im := pi / 2.0
      end
    else
      begin
        v.re := mag(u);   v.im := arctan(u.im / u.re)
      end
  end { polar };


procedure cln(z: complex; var c: complex);
{ Natural logarithm of a complex }

  var
    p: complex;

  begin
    polar(z, p);   c.re := ln(p.re);   c.im := p.im
  end { cln };
```

```
function order(z: complex): integer;
{ Gives a starting and even order for recursive computation }

  var
    a: real;
    m: integer;

  begin
    a := mag(z);
    if a < 0.1   then m := 10
    else
      begin if a < 2.0   then m := 28   else m := round(1.2 * a + 48.0)
      end;
    order := m;   if odd(m)   then order := m + 1
  end { order };


procedure sign(u: complex; var v: complex);
{ Changes the sign of a complex }

  begin
    v.re := - u.re;   v.im := - u.im
  end { sign };


procedure check(z: complex);
{ Checks to see if the function argument is outside range }

  var
    a, b: real;

  begin
    a := abs(z.re);   b := abs(z.im);
    if ((a < 1.0e - 5) and (b < 1.0e - 5)) or ((b <> 0.0) and (b < 1.0e
      - 5))
    then
      begin
        write('W: small argument which causes exponent error = ');
        cwrite(z);   stop;
      { Exit to terminate program }
      end;
    if b > 50.0 then
      begin
        write('W: argument with imaginary part outside range = ');
        cwrite(z);   stop;
      { Exit to terminate program }
      end
  end { check };


begin { Bessel2 }
  read(p);   n := abs(p);
  if n >= lim then
    begin
      writeln('W: required order ', p: 6, ' is outside the range (', -
         max: 4, ',', max: 4, ')');
      stop;
    { Exit to terminate program }
    end;
  cread(z);   check(z);
{ If z is outside range, exit to terminate program }
  m := order(z);
  if m >= lim then
    begin
      writeln('W: starting order ', m: 6,
          ' exceeds the specified maximum', max: 4);
      stop;
    { Exit to terminate program }
    end;
  zero.re := 0.0;   zero.im := 0.0;   sum := zero;   esum := zero;
  osum := zero;   f[m + 1] := zero;   f[m].re := 1.0e - 30;
  f[m].im := 0.0;
  for i := m downto 1 do
    begin
      quotient(f[i], z, w);   mult(2.0 * i, w, w);
      sub(w, f[i + 1], f[i - 1])
    end;
  k := m div 2;
  if abs(z.re) > 10.0 * abs(z.im)
  then
    begin
      for i := 1 to k do add(sum, f[2 * i], sum);   mult(2.0, sum, sum);
      add(sum, f[0], norm)

    end
  else
    begin
      for i := 1 to k do
        begin
          if odd(i)   then add(osum, f[2 * i], osum)
          else add(esum, f[2 * i], esum)
        end;
      sub(esum, osum, sum);   mult(2.0, sum, sum);
      add(sum, f[0], sum);   ccos(z, u);   quotient(sum, u, norm)
    end;
  for i := 0 to m do quotient(f[i], norm, f[i]);
{ Bessel functions of 1st kind }
  esum := zero;   osum := zero;   l := 1;
  if n = 0
  then
    begin { Yo }
      for i := 1 to k do
        begin
          l := - l;   mult(l / i, f[2 * i], u);   add(esum, u, esum)
        end;
      mult(2.0, esum, esum);   mult(0.5, z, u);   cln(u, u);
      u.re := u.re + euler;   product(u, f[0], u);   sub(u, esum, u);
      mult(tpi, u, yo);   fn := yo;   writeln;   writeln;
      write('     Function argument = ');   cwrite(z);   writeln;
      write('     Bessel function of the second kind and order 0 = ');
      cwrite(fn);   writeln;   writeln;   stop;
    { Exit to terminate program }
    end { Yo }
  else
    begin { Yn where n <> 0 }
      for i := 1 to k do
        begin
          l := - l;   mult(l / i, f[2 * i], u);   add(esum, u, esum);
          sub(f[2 * i - 1], f[2 * i + 1], v);   mult(l / i, v, v);
          add(osum, v, osum);
        end;
      mult(2.0, esum, sum);   mult(0.5, z, u);   cln(u, u);
      u.re := u.re + euler;   product(u, f[0], v);   sub(v, esum, v);
      mult(tpi, v, yo);
    { Yo }   product(u, f[i], v);
      quotient(f[0], z, w);   sub(v, w, w);   add(w, osum, w);
      mult(tpi, w, y1);
    { Y1 }   i := 1;
      while i < n do { Forward recursion }
        begin
          quotient(y1, z, u);   mult(2 * i, u, u);   sub(u, yo, y2);
          yo := y1;   y1 := y2;   i := i + 1;
        end;
    { Forward recursion }
      fn := y1;   if (p < 0) and odd(p)   then sign(fn, fn);   writeln;
      writeln;   write('     Function argument = ');   cwrite(z);
      writeln;
      write('     Bessel function of the second kind and order ', p: 4,
         ' = ');
      cwrite(fn);   writeln;   writeln
    end;
{ Yn }
1:
end { bessel2 }.
```

```
{* Purpose:
     Library routines to manipulate character strings in Pascal.

 * Author:
     Judy M. Bishop, Computer Science Division,  University of the
     Witwatersrand,  Johannesburg 2001, South Africa.

 * Description of routines:
     StringInitialize -- set up  the free space list ... called first
                         and once.
     StringError      -- Internal error reporting routine.
     News             -- Internal string allocation routine.
     Disposes         -- Internal string deallocation routine.
     Rewrites         -- User callable. Initialize a string for writing.
     Resets           -- User callable. Initialize a string for reading.
     Length           -- User callable function.
                           Returns string's length.
     Eofs             -- User callable function.
                           True if at end of string.
     Puts             -- Internal string character put routine.
     Gets             -- Internal string character get routine.
     Opens            -- User callable string creation routine.
     Closes           -- User callable string removal routine.
     Reads            -- User callable read string routine.
     Writes           -- User callable write string routine.
     Suppress         -- User callable trailing blank removal routine.
     Assign           -- User callable string assignment routine.
     Compare          -- User callable function returning the
                           relationship between two strings.
     AlfaToString     -- User callable assignment of alfa to string.
     CharToString     -- User callable assignment of char to string.

  An implementation of character string primitives using Pascal's
  dynamic storage allocation facilities.  The routines follow Arthur
  Sale's recommendation that strings be treated as sequences of
  characters.  Pascal sequences are processed by file routines, thus
  these string routines use similar names for similar functions.

 * Computer System:
     IBM 360/370 AABC Pascal compiler version 1.2.

 * References:
     J. M. Bishop, 'Implementing Strings in Pascal', "Software -
     Practice and Experience", 9(9), 779-788 (1979).
     A. H. J. Sale, 'Strings and the sequence abstraction in Pascal',
     "Software - Practice and Experience", 9(8), 671-683 (1979).
 *}


program stg(input, output);

const
  chunksize = 32;
  alfalen = 10;

type
        natural = 0 .. maxint;
            text= file of char;
            alfa= packed array [1 .. alfalen] of char;
       chunkptr = ^chunk;
          chunk = record
                              next: chunkptr;
                              line: packed array [1 .. chunksize] of
                                    char
                  end;
         string = record
                                 w: char;
                            length: natural;
                          position: 0 .. chunksize;
                             start,
                           current: chunkptr;
                           chunkno: natural;
                            status: (reading, writing, notready)
                  end;
       relation = (before, beforeorequalto, equalto, afterorequalto,
                   after, notequalto);

var
           avail: chunkptr;

procedure stringinitialize;

  begin avail := nil; end;

procedure stringerror(n: natural);

  begin
    writeln;   writeln(' **** execution error in string library ****');
    case n of
      1: write(' put attempted in read state ');
      2: write(' get attempted in write state ');
      3: write(' get attempted beyond end of string ');
      4: write(' delete portion bigger than string ');
      5: write(' extract portion bigger than string ');
      6: write(' inserting beyond end of string ')
    end;
    writeln(' ****');
{}  halt
  end { stringerror };

procedure news(var p: chunkptr);

  var
                  i: 1 .. chunksize;

  begin
    if avail = nil
    then
      begin
        new(p);   with p^ do for i := 1 to chunksize do line[i] := ' ';
        { undefined }
      end
    else begin p := avail;   avail := avail^.next end;
  end { news };

procedure disposes(p: chunkptr);

  begin p^.next := avail;   avail := p; end;

procedure rewrites(var s: string);

  begin
    with s do
      begin
        if start = nil
        then begin news(start);   start^.next := nil; end;
        current := start;   position := 0;   chunkno := 0;
        length := 0;   status := writing
      end
  end { rewrites };

procedure resets(var s: string);

  var
                c: chunkptr;

  begin
    with s do
      begin
        if status = writing
        then
          begin
            length := length + position;   c := current^.next;
            current^.next := nil;
            while c <> nil do
              begin current := c^.next;   disposes(c);   c := current
              end
          end;
        current := start;   position := 1;   chunkno := 0;
        status := reading;
        if current <> nil   then w := current^.line[1]   else w := ' ';
      {  when reset done on an empty string  }
      end
  end { resets };

function length(s: string): natural;

  begin resets(s);   length := s.length; end;

function eofs(s: string): boolean;

  begin
    with s do eofs := (length + 1) = chunkno * chunksize + position;
  end { eofs };

procedure puts(var s: string);

  begin
    with s do
      begin
        if status = reading   then stringerror(1);
        if position = chunksize
        then
          begin
            if current^.next = nil then
              begin
                news(current^.next);   current^.next^.next := nil;
              end;
            current := current^.next;   chunkno := chunkno + 1;
            length := length + chunksize;   position := 1;
          end
        else position := position + 1;
        current^.line[position] := w;   w := ' ';
      end
  end { puts };

procedure gets(var s: string);

  begin
    with s do
      begin
        if status = writing   then stringerror(2);
        if eofs(s)   then stringerror(3);
        if position = chunksize
        then
          begin
            current := current^.next;   chunkno := chunkno + 1;
            position := 1
          end
        else position := position + 1;
        if current <> nil   then w := current^.line[position]
        else w := ' ';
      { when the eof coincides with the end of a chunk. }
      end
  end { gets };

procedure opens(var s: string);

  begin
    with s do
      begin
        length := 0;   chunkno := 0;   position := 0;   start := nil;
        current := nil;   status := notready;   w := ' ';
      end
  end { opens };

procedure closes(var s: string);

  begin
    with s do
```

```
      while start <> nil do
        begin
          current := start^.next;   disposes(start);
          start := current
        end;
  end { closes };

procedure reads(var from: text; var s: string);
{  reads until an end-of-line.  }

  begin
    rewrites(s);   if eoln(from)   then get(from);
    while not eoln(from) do
      begin s.w := from^;   puts(s);   get(from); end;
  end { reads };

procedure writes(var onto: text; s: string);

  begin
    resets(s);
    while not eofs(s) do begin write(onto, s.w);   gets(s); end
  end { writes };

procedure suppress(var s: string);
{  removes trailing blanks.  }

  const
    space = ' ';

  var
            spaces: boolean;
              mark,
                 i,
                 l: natural;

  begin
    l := length(s);   mark := 0;   resets(s);   spaces := false;
    for i := 1 to l do
      begin
        if s.w = space
        then
          begin
            if not spaces   then begin spaces := true;   mark := i end
          end
        else begin spaces := false;   mark := 0; end;
        gets(s)
      end;
    if mark > 0   then s.length := mark - 1;   resets(s);
  end { suppress };

procedure assign(var s1: string; s2: string);

  begin
    rewrites(s1);   resets(s2);
    while not eofs(s2) do
      begin s1.w := s2.w;   puts(s1);   gets(s2); end;
  end { assign };

function compare(s1: string; r: relation; s2: string): boolean;

  var
              less,
            equal: boolean;
              ls1,
              ls2: natural;

  begin
    ls1 := length(s1);   ls2 := length(s2);   resets(s1);   resets(s2);
    equal := ls1 = ls2;   less := false;
    while (equal and not less) and not eofs(s1) and not eofs(s2) do
      begin
        equal := s1.w = s2.w;   less := s1.w < s2.w;   gets(s1);
        gets(s2)
      end;
    case r of
      before: compare := less;
      beforeorequalto: compare := less or equal;
      equalto: compare := equal;
      afterorequalto: compare := not less or equal;
      after: compare := not less;
      notequalto: compare := not equal
    end;
  end { compare };

procedure alfatostring(a: alfa; var s: string);

  const
    space = ' ';

  var
                i: natural;
            state: (scanning, ended, spacefound);

  begin
    rewrites(s);   i := 1;   state := scanning;
    repeat
      if i > alfalen   then state := ended
      else
        if a[i] = space   then state := spacefound
        else begin s.w := a[i];   puts(s);   i := i + 1 end
    until state <> scanning;
  end { alfatostring };

procedure chartostring(c: char; var s: string);

  begin rewrites(s);   s.w := c;   puts(s) end;

begin end.
```

Articles

CONFORMANT ARRAYS IN PASCAL

by A.H.J.Sale
University of Tasmania
(at the request of Andy Mickel)

1. CONFORMANT ARRAYS AND THE NEW STANDARD

The draft proposal for an ISO Standard for Pascal contains within it a definition of what I shall call a "conformant array parameter". The basic concept is that of a parameter specification which allows a formal-parameter to assume the values and types of different actual array-parameters.

How did the draft Standard acquire this feature? And why?

2. PRESSURE GROUPS

During the preparation of the draft Standard, a considerable amount of public comment was received by the sponsoring body, BSI, and the chairman of its Pascal Committee, Tony Addyman. (I seem to recall a figure of 10kg.) A significant amount of this was devoted to the problem of writing general procedures to sort and perform other array operations, inevitably leading either to suggestions of a full dynamic array facility, or some sort of conformant array parameter.

Of course, contributors to Pascal News have not been idle in this regard either. Many suggestions for conformant array parameters have been received; some good, some not. It is clear that this is perceived by many to be a deficiency in the language, though there are quite good arguments to support the view that it is only a deficiency viewed in a particular way. Correct or not, the perception has led to pressure being applied to the Pascal Committee to put a feature of this sort in the draft, the Numerical Algorithms Group (NAG) at Oxford being an important example.

However, this pressure had not had an effect by the time of the publication of the third Working Draft (N462) widely published last year. Then, two critical pressures were applied to the Committee by N. Wirth and C.A.R.Hoare (independently) supporting the view that now was the time to add a conformant array feature to Pascal. It seems safe to assume that in the absence of pressure from such quarters the urge to add to Pascal would have been successfully resisted by BSI.

3. PROPOSALS

The proposals put forward by way of defining a conformant array feature have been many and varied. Some have been strange in their exploitation of minor aspects of Pascal, and many others have been obsessed by syntax to the exclusion of what the construct should mean. It is quite clear, even before you look seriously, that the addition of conformant arrays to Pascal is not a trivial task.

The BSI Pascal Committee accordingly had to choose something to satisfy the pressures from the joint designers of the language. They rejected the silly suggestions of course, and chose to put in the document which went to

Turin (N510) a considerably modified version of a scheme which seemed to originate with Jacobi. Subsequently, it became clear that there were better possibilities, and BSI withdrew support for its own draft, in favour of an improved one, now incorporated in the Draft Proposal. This scheme, which seems to have originated with N.Wirth, has been examined by both opponents and proponents of the addition in order to ensure that at least if there is to be an addition, it should be the best one possible. That is my own position.

The key idea behind the current proposal is that it preserves the abstraction of an array as a complete mapping, and incorporates a number of "compile-time" checks on the validity of actual calls. The cost is that of introducing what the draft proposal calls a "schema"; or in other words a specification which is not a type but a rule for identifying and constraining a set of types. Thus the type of a formal conformant-array-parameter is not known from its declaration, but is supplied by each call. The consequences are very simple outside this one point, especially in defining parameter-list congruity which many other proposals make very heavy weather of indeed.

4. TURIN

At Turin, the site of the very first computer conference ever, there was a considerable amount of discussion of the conformant array proposal. Opposition to the proposal was stated by the US, and one or two other people, but there was clearly a substantial majority which would accept the inclusion of such a feature, and many indeed welcomed it. Consequently, the feeling of the experts group was recorded as being in favour of some form of conformant array parameter being in the first Standard.

Discussion then turned on the form of the parameter mechanism, with the possibilities being the BSI original, the redraft now incorporated, and an improved Jacobi-like proposal. Conformant array parameters took over two hours of technical discussion (about 12% of the total), and also ran into dinner, breakfast and a coffee-break. However, it is useful to realize that the Turin meeting perceived this as an important issue, but not of over-riding importance.

5. TIMELINESS

Part of the pressure to make this feature appear in the Draft Standard arises from a desire to have important numerical algorithms translated into Pascal, and the language used in this area now dominated by Fortran. But simply because this pressure is present, many implementors have already inserted a feature of this general type into their implementations, and they differ very widely. Not surprizingly, not many implementors think much about the abstractions behind their extensions, or perhaps they borrow extensions. The signs are there that if conformant array parameters are not standardized now, they may as well never be for all the good it will do.

Speaking personally, I had had six new implementors call me in the last month, and all of them have asked for guidance on how they should implement conformant array parameters. Such interest by new commercial implementations is significant; however the existing implementations are likely to be harder to bring into any sort of conformance.

Reluctantly, because I was not an original supporter of conformant arrays, I have been convinced that both timeliness and utility require the action that was taken at Turin. I think the inclusion is warranted.

6. CURRENT STATUS

To keep readers of Pascal News informed, I reproduce some pieces of the draft proposal as they relate to conformant array parameters. It can be seen that the addition is entirely localized within the parameter list, except for the addition of one item to 'factor' (and no need even to write anything about it in the accompanying text). The conformant array parameter schema is well-crafted so that it hangs together as an integrated whole, and the reasons for most of the statements will be clear after some thought.

The exact syntax may be changed without damage to the proposal. The use of "..", ":" and ";" is based on analogies with subranges, variable-declarations, and formal parameter lists respectively. Other people may prefer to use commas or whatever. It doesn't really matter as long as the abstraction is right, except for students.

7. IMPLEMENTATION

I have noticed some people saying that the implementation of conformant arrays is unproven, and I should like to sharply disagree. There is no problem whatsoever about the implementation of any of these schemes, and they have been well-known for a very long time. The whole argument has been around fitting the idea into Pascal with the minimum of change to its fabric. Any competent implementor will be able to implement this feature on any machine I know, and existing implementations which differ can be altered very easily.

There is one exception. Not that it is unknown, but that we know very well that if we are going to allow packed arrays to be actual parameters to a conformant array parameter, then we will be forced into either giving up packing completely on some machines, or imposing some ugly restrictions on conformant array parameters, or passing some bit-size argument and requiring the called procedure to reproduce the vagaries of the packing algorithm. The problem is essentially that the size (in bits, say) of the component-type may not be known until execution. For this reason, the use of packed in a conformant array parameter was not allowed.

It should be realized that the inclusion of packed in the Standard means that all implementors must provide it (do not fall into the trap of thinking of the Standard as a permissive one or a layered one such as COBOL), and the likely effects are simply to cause it to be ignored and the effectiveness of the Standard nullified, or to cause no packing to take place when the 'Standard' compiler option is set. This would be singularly unfortunate for a feature whose main use seems to be to simulate something else (strings). It should be pointed out that its exclusion means that some implementors may choose to provide it as an extension. The abstract meaning is clear; the syntax is clear; only the implementation is difficult.

Arthur Sale

Section 6.6.3

```
variable-parameter-specification =
        "var" identifier-list ":"
        (type-identifier | conformant-array-schema) .
conformant-array-schema =
        "array" "[" index-type-specification
        { ";" index-type-specification } "]" "of"
        ( type-identifier | conformant-array-schema ) .
index-type-specification =
        bound-identifier ".." bound-identifier
        ":" ordinal-type-identifier .
bound-identifier = identifier .
```

The occurrence of an identifier within an identifier-list of a value-parameter-specification or a variable-parameter-specification shall be its defining-point as a parameter-identifier for the region that is the formal-parameter-list in which it occurs and its defining-point as a variable-identifier for the region that is the procedure-block or function-block, if any, whose formal parameters are defined by that formal-parameter-list.

The occurrence of an identifier as a bound-identifier within an index-type-specification shall be its defining-point as a bound-identifier for the region that is the formal-parameter-list in which it occurs and for the region that is the procedure-block or function-block, if any, whose formal parameters are defined by that formal-parameter-list.

If the component of a conformant-array-schema is itself a conformant-array-schema, then an abbreviated form of defintiion may be used. In the abbreviated form, all the index-type-specifications shall be contained within the same enclosing square brackets, a single semi-colon replacing each sequence of right-square-bracket "of" "array" left-square-bracket that occurred in the full form. The abbreviated form shall be equivalent to the full form.

Examples:

```
array[u..v: T1] of array[j..k: T2] of T3
array[u..v: T1; j..k: T2] of T3
```

6.6.3.3 Variable parameters. The actual-parameter (see 6.7.3 and 6.8.2.3) corresponding to formal parameters that occur in the same identifier-list in the formal-parameter-list shall all be of the same type. This type shall be the same as the type of the type-identifier in the variable-parameter-specification if the formal parameter is so specified, otherwise it shall be conformable to the conformant-array-schema in the variable-parameter-specification. Each formal parameter shall denote the corresponding actual-parameter during the entire activation of the block. Any operation involving the formal parameter shall be performed immediately on the actual-parameter.

If access to the actual-parameter involves the indexing of an array and/or the selection of a field within a variant of a record and/or the de-referencing of a pointer and/or a reference to a buffer-variable, these actions shall be executed before the activation of the block.

Components of variables of any type designated packed shall not be used as actual variable parameters.

If T1 is an array-type, and T2 is the type the ordinal-type-identifier of a conformant-array-schema, then T1 is conformable with T2 if all the following four statements are true.

(a) The index-type of T1 is compatible with T2.
(b) The smallest and largest value of the index-type of T1 lie within the closed interval defined by values of T2.
(c) The component-type of T1 is the same as a component-type of the conformant-array-schema, or is conformable to a component conformant-array-schema.
(d) T1 is not designated packed.

It shall be an error if the smallest or largest value of the index-type of T1 lies outside the closed interval defined by the values of T2.

During the entire activation of the block, the first bound-identifier shall denote the smallest value of the index-type of the actual-parameters, and the second bound-identifier shall denote the largest value of the index-type of the actual-parameters.

6.6.3.6 Parameter list congruity. Two formal-parameter-lists shall be congruous if they contain the same number of parameters and if the parameters in corresponding positions match. Two parameters shall match if any of the four statements that follow is true.

(a) They are both value parameters of the same type.
(b) They are both variable parameters of the same type, or have equivalent conformant-array-schemas. Two conformant-array-schemas are equivalent if they have the same ordinal-type specified in their index-type-specifications and their components are either of the same type or are equivalent conformant-array-schemas.
(c) They are both procedural parameters with congruous parameter lists, if any.
(d) They are both functional parameters with congruous parameter lists, if any, and the same result-type.

Section 6.7.1

```
factor = variable | unsigned-constant | bound-identifier |
         function-designator | set-constructor |
         "(" expression ")" | "not" factor .
```

DEPARTMENT OF THE ARMY
USA DARCOM AUTOMATED LOGISTICS MANAGEMENT SYSTEMS ACTIVITY
PO BOX 1578, ST LOUIS, MISSOURI 63188

DRXAL-T

18 January 1979

Mr. Andy Mickel
Pascal User's Group
University Computer Center: 227 EX
208 SE Union Street
University of Minnesota
Minneapolis, MN 55455

Dear Andy:

Our agency sent questionnaires to about 950 members of the Pascal User's Group in the United States in order to gather information on their experience with the language and available software. Thank you for providing us with a copy of the User's Group mailing list for this endeavor.

We are submitting the attached copy of the results of our survey to you for publication in the Pascal News. Also, enclosed is a copy of the questionnaire for your information. If you have any questions, please contact John McCandliss, 314-268-2786, or Sue Burklund, 314-268-5151.

1 Incl
As stated

ROBERT R. RANSOM
Director for ADP Technology

PASCAL SURVEY

Pascal is a computer language developed by Niklaus Wirth at ETH in Zurich, Switzerland. It is derived from Algol 60, but is more powerful and incorporates structured programming principles. Pascal has been implemented on a variety of computers throughout the world with the most common being Control Data Corporation and Digital Equipment Corporation computers. Its widest use to date has been as an instructional tool to teach students the principles of programming in a structured manner, but some computer companies, notably CDC and Texas Instruments are using it as a systems programming language.

ALMSA developed a questionnaire which was sent to approximately 950 members of the Pascal User's Group in the United States. We received about 120 usable responses, which were analyzed to provide the statistics for this report. The responses, especially in the area of relative speed and size of Pascal generated code compared to other languages, were often incomplete, so each area of the report indicates the number of responses on which it is based.

The questionnaire brought some interesting facts about Pascal usage to light. The first interesting statistic is that almost ½ of the responses were from educational institutions, and another ¼ were from computer companies. Most of the government organizations responding were research oriented. It is safe to say that as yet, Pascal has not moved into the mainstream of computer programming, although judging by the fact that over 4/5 of the respondents said that Pascal usage at their installation was increasing this development might be forthcoming in the future.

Another interesting fact is that 3/5 of the respondents were using Standard Pascal. Pascal was highly rated as an educational tool, but got its lowest ratings as a language for writing operating systems and business applications. Extensions of Pascal, such as Brinch Hansen's Concurrent Pascal, will be necessary before Pascal will be acceptable for writing operating systems. Other extensions, such as better I/O capabilities will be necessary to make Pascal an acceptable business programming language.

It is hard to make any judgment as to the efficiency of Pascal generated code, because of the small number of responses, and the large variety of compilers cited. In most cases, the Pascal generated code was both slower and larger compared to modules in assembly language and other high level languages. However, a couple of compilers, including the widely used University of Colorado version, were producing code that was compared favorably with that produced by FORTRAN compilers.

PASCAL QUESTIONNAIRE STATISTICS

General Statistics:	Number	Percent
Number of questionnaires mailed	950	100%
Number of replies received	155	16%
Replies from organizations which didn't have working compilers or said they couldn't answer our survey	33	3%
Usable replies	122	13%

Types of Respondents:			
a.	Governmental organizations	10	8.2%
b.	Educational organizations	60	49.2%
c.	Business organizations	23	18.9%
d.	Computer organizations	29	23.8%
	Total	122	100.1%

Type of Pascal Used:			
a.	Standard	78	65.0%
b.	Subset of standard	12	10.0%
c.	Sequential Pascal	11	9.2%
d.	Concurrent Pascal	5	4.2%
e.	Other	14	11.7%
	Total	120	100.1%

Note: These numbers are not exact since some organizations had more than one Pascal compiler.

How many of these organizations use Pascal compilers as opposed to interpreters?

a.	Pascal compilers	87	76%
b.	Pascal interpreters	15	13%
c.	Both	13	11%
	Total	115	100%

Percentage of coding being done at each installation in Pascal:

a.	Number of replies	94
b.	Average % of coding	14.5%

Trend of Pascal usage at each installation:

a.	Replies	116
b.	Increasing	84%
c.	Decreasing or stable	16%

Note: The following three areas were rated on a 0 to 3 scale where:
0 = Poor
1 = Adequate
2 = Good
3 = Excellent

	Number of replies	Average Rating
Reliability of Pascal compilers:	116	2.2
Suitability for the following applications:		
a. FORTRAN replacement	110	2.1
b. ALGOL replacement	95	2.4
c. Educational use	104	2.6
d. Operating systems	88	1.4
e. Systems programming	101	2.0
f. Business applications	87	1.4
g. Scientific applications	99	2.1
Pascal's capabilities in various programming areas:		
a. I/O operations	114	1.4
b. Numeric computations	122	1.8
c. Integer arithmetic	111	2.4
d. Character handling	114	1.9
e. String handling	112	1.1

Speed/size of Pascal generated code compared to a similar module on the same system in another language:

Speed			Size		
a.	Number of replies	20	a.	Number of replies	13
b.	Faster	3	b.	Smaller	1
c.	Slower	17	c.	Larger	12

Comments that many respondents made about the limitations of Pascal and what they thought would be the most useful extensions to Pascal:

a. Formatted I/O
b. Random access capabilities
c. Better interfaces with other programs
d. Ability to initialize variables
e. Bit strings
f. Make it easier to compile procedures separately
g. More interactive functions
h. Dynamic arrays

CONVERTING AN APPLICATION PROGRAM FROM OMSI PASCAL 1.1F TO AAEC PASCAL 8000/1.2

Geoffrey R Grinton
State Electricity Commission of Victoria
Richmond, Victoria 3121, Australia

I recently had occasion to transfer an application program originally written on a PDP 11/34 system using RT-11 and OMSI Pascal 1.1F to an installation running AAEC Pascal 8000/1.2 under MVS on a dual IBM 370.

Although the program had originally been written with this transfer in mind, and hence with a minimum of system dependent features, there were several areas in which unexpected changes had to be made. Some of the changes are of a trivial nature, and were expected. Others, however, were less obvious, and posed some problems.

This note describes the differences encountered, and is intended to show others the sorts of problems likely to be encountered in such an exercise.

1. The original version was written using a mixture of upper and lower case characters. When this was fed into the AAEC compiler the compiler crashed; no indication of the likely cause of the problem was given, so a bit of inspired guess-work was required. The solution used was to change the whole program to upper case.

2. It was necessary to convert occurences of the characters [,] and ^ to the AAEC equivalents, namely (., .) and @. I have since found that the AAEC compiler accepts [and], but this is not documented.

3. There were several occurences of VALUE as a variable name. Since the AAEC compiler allows a VALUE segment, which follows immediately after the VAR segment, this caused it some confusion.

4. I had omitted to include names of external files, including INPUT and OUTPUT, in the program header (which is optional in the OMSI compiler), so these had to be inserted.

5. It was necessary to reduce the nesting level of procedures, since AAEC allow only six levels. The OMSI compiler allows up to ten levels. Such a restriction would appear to me to be contrary to the philosophy of structured programming, as it requires the programmer to either use larger (and hence less comprehensible) blocks, or to place procedures which should logically be contained in another block at a higher level.

6. The OMSI system had failed to detect an invalid assignment to a subrange variable. This was correctly diagnosed by the AAEC run-time system. The particular example was a subtle form of:

```
var index : 1..top;
  ...
index := 0;
```

7. The AAEC system, when running under the Time Sharing Option (TSO) of MVS does not actually write to a terminal until a line is completed, with writeln. Hence all prompting messages had to be changed to use writeln instead of write.

8. It was necessary to change all output formats to allow for a carriage control character. This was not strictly necessary, but it was required if the system default DCB information was to be used (ie RECFM=FA).

9. Since the AAEC version does not specifically allow for interactive use, all input had to be changed so that the file pointer was always defined. This was done primarily by changing all occurences of readln(..) to readln; read(..), although several other minor programming changes were also necessary.

10. The OMSI compiler does not pre-declare files INPUT and OUTPUT, and consequently does not allow references to input^ to look-ahead on the input file. With the changes described in point 9, it was useful to be able to do this in the AAEC verion of the program. Further changes became necessary, however, when I realised that the system was adding extra blanks to the ends of my input lines, to fill them out to 80 characters. (I can't say that I wasn't warned by Jensen and Wirth, but that one took a lot of finding!)

11. OMSI Pascal uses modified forms of reset and rewrite to attach actual RT-11 files to internal file variables. The AAEC system requires this connection to be made externally, and hence the appropriate initialisation routine had to be changed.

12. As OMSI Pascal ignores the 'packed' attribute, and automatically packs all character arrays and strings, I had not specified arrays of type char as packed. This was necessary on the AAEC system for proper operation of my program.

The conversion process was, despite the differences outlined above, probably simpler than I had expected. Apart from the I/O related difficulties, there were few incompatibilities between the systems, and conversion of the whole program of 1200 lines was completed within a couple of days.

15th May, 1979

DOES SCOPE = BLOCK IN PASCAL?

T. P. Baker*
Department of Computer Science
The University of Iowa
Iowa City, Iowa 52242

and

A. C. Fleck
Department of Computer Science
and
Weeg Computing Center
The University of Iowa
Iowa City, Iowa 52242

INTRODUCTION

There seems to have developed some controversy over whether the scopes of identifiers are (or should be) synonymous with blocks in PASCAL. In this note we call attention to the formal statement of the "rules" dealing with this situation, point out several other items in the literature that address the question of the title, and present our own personal conclusions. We relate our comments first with respect to "Standard" PASCAL and then to the new BSI/ISO Working Draft Standard PASCAL.

WIRTH'S STANDARD PASCAL

There are several levels of documentation to consider in this case, in decreasing order of abstraction: the Report [2], the User Manual [2], and the several E.T.H. compilers. Arthur Sale in [3] argues strongly the position that scope = block. But we would like to suggest that there are loopholes. The Report is unfortunately vague. In section 10, we are told that scope = procedure (or function) declaration and that identifiers are not known outside their scope. But it gives no details of how they are known inside their scope. The crucial issue is nested scopes which are mentioned in Section 2 but for which no rules are given. Section 4 of the Report tells us that the association of an identifier must be unique within its scope. This is essentially the extent of the specifications in the Report. In this light, consider the following example:

```
1  PROGRAM P1(OUTPUT);
2       PROCEDURE Q; BEGIN WRITELN(1) END;
3       PROCEDURE R;
4            PROCEDURE S; BEGIN Q END;
5            PROCEDURE Q; BEGIN WRITELN(2) END;
6       BEGIN S END;
7  BEGIN R END.
```

Now there are two definitions provided for identifier 'Q' within nested scopes. The one within R must not be known outside R. There is only one invoking instance of the identifier 'Q' (hence its association must be unique) and its occurrence is validly within both scopes and the Report's rules give us no reason for preference.

---

*Present address: Mathematics Department, Florida State Univ., Tallahassee, FL 32306

Next we consider the User Manual. Here in Chapter 1 (pp. 6-7) we find it again stated that scope = procedure declaration. Also it is stated "the scope or range of validity of an identifier x is the entire block in which x is defined, including those blocks defined in the same block as x." Applied to program P1 above, this would seem to imply that the correct output of P1 is 1. However the above quote has a parenthetical comment that all identifiers must be distinct for this to apply and refers to Section 3.E for the case where identifiers are not necessarily distinct (this is the case with P1). Reading Section 3.E, we find that the definition of a variable definition in an inner block is valid throughout that block. This might suggest the correct output of P1 is 2. Actually this rule has nothing to do with program P1 as it deals exclusively with variable identifiers, the topic of Section 3.E. Unfortunately the other sections on type identifiers, procedure identifiers and constant identifiers give no rules at all.

The last, most specific and least satisfactory source for a resolution of scope rules (other than for variable identifiers) is the E.T.H. compilers. Because of Wirth's close association here, their performance must be considered significant. The output of both the Version 2 and Version 3 compiler for P1 is 1. This performance is supported by the rule in Chapter 1 (p. 8, item 16) of the User Manual that "All objects must be declared before they are referenced" (two exceptions noted are pointer types and forward procedures). In the absence of other rules about scope it is not unnatural to apply this one, hence accepting the outer definition throughout its scope until another occurs (the Version 2 and 3 compilers do violate the unique association rule which does not come up in P1). This is presumably the reason for Watt's [4] assumption that Sale [3] criticizes.

THE BSI/ISO STANDARD

We now turn our attention to the new Draft Standard [1]. While there are problems with the existing language specification, it is this new definition which causes us the most serious concern. The Draft Standard eliminates the previously existing omissions on the specification of scope rules. There is an explicit enumeration of the nested scope rules for all varieties of identifiers (see Section 6.2.1). Unfortunately, as we shall see, these rules imply that scope ≠ block for all cases except variable and type identifiers.

Each identifier has a defining occurrence and each defining occurrence has a scope which encloses all "corresponding occurrences" (a term not defined). Here the Draft Standard leaves some ambiguity as it does not state precisely where such scope begins and ends. Since the scope must enclose all "corresponding occurrences" we shall simply assume that the scope ends with the end of the block in whose heading the defining occurrence appears. The choice for the beginning of the scope is another question. Since each defining occurrence is prescribed as having a scope associated with it (i.e., scopes are associated with defining occurrences not blocks), one seems naturally forced to assume that such a scope begins with the defining occurrence. This assumption seems reinforced by the rule (in Section 6.4) that the scope of the defining occurrence of a type identifier does not include its own definition, except for pointer types. There is one exception to this assumption explicitly stated in rule (5) of Section 6.2.1. This rule states that the defining occurrence of any identifier or label must precede all its "corresponding occurrences" except for a pointer-type identifier which may have its defining occurrence anywhere in the type-definition part. Hence we assume that the scope of a pointer-type identifier begins with the beginning of the type-definition part rather than with its defining occurrence.

Now consider the previously given program example P1. There is no longer any doubt over what its correct output must be. This program has two defining occurrences of the identifier 'Q' (the specification of a defining occurrence for a procedure identifier is given in Section 6.6.1), in lines 2 and 5. The scope of the first extends to the end of P1 (i.e., lines 2-7) and the nested scope of the second extends to the end of procedure R (i.e., lines 5-6). Clearly then the call in line 4 is a "corresponding occurrence" for the definition in line 2, an association clearly violating ALGOL60-style scope rules.

The same situation prevails for constant identifiers. As an example consider

```
1  PROGRAM P2(OUTPUT);
2  CONST TWO = 2;
3        PROCEDURE Q;
4        CONST ONE = TWO;
5              TWO = 1;
6        BEGIN WRITELN(ONE) END;
7  BEGIN Q END.
```

We do not include the scope analysis for this program as it is similar to that for program P1. The upshot is the same as for procedure identifiers, namely scope ≠ block for constant identifiers.

On the other hand since type-identifiers cannot occur in a heading prior to the type-definition part, rule (5) of Section 6.2.1 implies that scope = block for type identifiers. For instance, in contrast to the previous examples, the program

```
1  PROGRAM P3(OUTPUT);
2  TYPE A = RECORD L : ↑A; C : REAL END;
3        PROCEDURE Q;
4        TYPE B = ↑A;
5              A = RECORD L : B; C : INTEGER END;
6        VAR X : B;
7        BEGIN NEW(X); X↑.C := 0.5 END;
8  BEGIN Q END.
```

is illegal because of the type conflict in the assignment in line 7 (however the Version 3 E.T.H. compiler finds it legal).

Also since variable identifiers cannot be used in the heading at all, these rules imply that scope = block for variable identifiers as well. Hence for the Draft Standard we get two answers to the question of the title; 'yes' for variable and type identifiers and 'no' for constant, procedure and enumeration-type identifiers.

CONCLUSIONS

The lack of specification of rules for nested scopes in the original PASCAL definition has resulted in different interpretations being taken by different implementations. This point has already been made in [5]. The fact that so basic an issue must be settled has been recognized in the development of a draft standard.

We feel that while the Draft Standard does resolve the ambiguities of scopes, the solution that is proposed is very poorly conceived. The answer to the question "does scope = block?" should be uniform for all varieties of identifiers and furthermore we agree with Sale [3], that uniform answer should be yes.

Programs P1 and P2 show how present scope rules provide for the binding of corresponding occurrences of identifiers to defining occurrences outside the block of the corresponding occurrence even though this block itself contains a defining occurrence. A convention which provides for the binding of one identifier to two definitions within the same block seems entirely contrary to the evolution of PASCAL.

The scope rules should state that the scope of a defining occurrence extends from the beginning of the block in whose heading it occurs to the end of this block. This would replace rules (1) and (2) of Section 6.2.1 of [1]. The other rules would be retained as stated; however we would rephrase rule (5) slightly to say that the completion of the definition for a defining occurrence must precede all corresponding occurrences—then the scope rule in Section 6.4 is dropped. This would make programs P1 and P2 illegal as they then violate rule (5)—the defining occurrence in the nested block does not precede first use. It has already been suggested [5] how this interpretation can be handled in a one-pass compiler. The only complication to this comes in the exception to rule (5) for pointer-types which must force the binding of all such identifiers (even those with definitions in enclosing scopes) to be deferred until the end of the type-definition part.

We feel the approach we suggest provides a conceptually cleaner solution to the scoping questions. The treatment of all varieties of identifiers is internally consistent and consistent with the conventions of other block structure languages as well. Moreover it conforms with the principle of locality. With the rules given in the present Draft Standard, a block can contain identifiers with both a local and a nonlocal binding—a very confusing situation.

REFERENCES

1. A.M. Addyman et al., "A draft description of PASCAL," Software-Pract. & Exper. 9,5(1979), 381-424; also PASCAL News 14(1979), 7-54.

2. K. Jensen & N. Wirth, PASCAL User Manual and Report, Springer-Verlag, Second Edition, 1975.

3. A. Sale, "Scope and PASCAL," SIGPLAN Notices 14,9(Sept. 1979), 61-63.

4. D.A. Watt, "An extended attribute grammar for PASCAL," SIGPLAN Notices 14,2(Feb. 1979), 60-74.

5. J. Welch, W.J. Sneeringer & C.A.R. Hoare, "Ambiguities and insecurities in PASCAL," Software-Pract. & Exper. 7(1977), 685-696.

A NOTE ON PASCAL SCOPES

T. P. Baker and A. C. Fleck

Department of Computer Science
The University of Iowa
Iowa City, Iowa 52242

In response to the recent efforts toward development of a PASCAL standard [1], we would like to point out a peculiarity we have observed in the PASCAL notion of scopes, as exemplified in the E.T.H. compilers, and to suggest how a "cleaner" alternative notion might be implemented.

Beginning with ALGOL60, "block structured" languages have followed the convention that scopes of local declarations correspond to the boundaries of the blocks in which they occur. Since PASCAL superficially appears to follow this convention, a programmer is likely to go along for some time before he stumbles upon a case where PASCAL scopes do not correspond to block boundaries. When he does, it is likely to be a source of confusion. For example, consider the programs and output below (from Version 3 of the PASCAL 6000 compiler):

```
1 PROGRAM P1(OUTPUT);
2    PROCEDURE Q; BEGIN WRITELN(1) END;
3    PROCEDURE R;
4       PROCEDURE S; BEGIN Q END;
5       PROCEDURE Q; BEGIN WRITELN(2) END;
6    BEGIN S; Q END;
7 BEGIN R END.

1
2
```

```
1 PROGRAM P2(OUTPUT);
2 TYPE A = CHAR;
3    PROCEDURE Q;
4    TYPE B = ^A;
5         A = RECORD L,R: B END;
6    VAR X: B;
7    BEGIN NEW(X); X^ := 'A' END;
8 BEGIN Q END.
```

```
1 PROGRAM P3(OUTPUT);
2 VAR F: INTEGER;
3    PROCEDURE Q;
4       PROCEDURE R; BEGIN WRITELN(F) END;
5       FUNCTION F: INTEGER; BEGIN F := 2 END;
6    BEGIN R; WRITELN(F) END;
7 BEGIN F := 1; Q END.

1
2
```

Note that according to current and proposed scope rules [1], this is the "correct" program behavior in each case.

We propose that PASCAL can be standardized to follow the ALGOL60 scope conventions, with the added restriction that (except in recursive pointer type declarations) no use of an identifier may precede its declaration (this appears to be the approach taken in ADA [2]). Thus, program P1 above would be considered incorrect, since the use of Q in procedure S precedes a local definition of Q. P3 would be incorrect for a similar reason, because the use of F in procedure R precedes a local declaration of F. Program P2 would be considered incorrect, but for a different reason. The variable X would be interpreted as a pointer to a record, so that the assignment "X^ := 'A'" would be a type conflict. This is exactly what would have happened if the outer declaration "A = CHAR" had not been present. In this case, the convention followed by the compiler not only makes the interpretation of the procedure Q dependent in an unobvious way on its global environment, but also effectively blocks the possibility of defining a pointer type for the local record type A.

A single pass compiler can enforce these conventions. On first encountering a use of an identifier X that is not yet declared in the local block, the compiler attempts to resolve the reference to a previously processed nonlocal declaration, say D, in one of the surrounding blocks. If this search is successful, the processor creates new "dummy" entries for X in the symbol table for the local block and all surrounding blocks, out to the block where D appeared. These dummy entries will include a pointer to the entry corresponding to D and will serve the purpose of insuring that any subsequent declaration of X locally will be deleted and treated as an error.

PASCAL already provides means for handling the few cases where forward references are unavoidable. For procedures, functions, and labels, there are forward declarations. For recursively defined pointer types, processing can be deferred until it can be determined whether a type identifier should be resolved as a local or nonlocal reference. For example, processing of "B = ^A" in P2 would be deferred until the local declaration of A was encountered (or until the end of the TYPE section).

We believe that the proposed conventions are an improvement in the direction of simplicity and conformity to established practice. Furthermore, as exemplified best in program P2, they improve program modularity, by permitting reliable local resolution of references, which under present rules is impossible.

[1] A.M. Addyman et al. "A draft description of PASCAL," Software Pract. & Exper. 9, 5(1979), 381-424; also PASCAL News 14(1979), 7-54.

[2] Preliminary ADA Reference Manual, SIGPLAN Notices 14, 6(1979).

AN ALTERNATE APPROACH TO TYPE EQUIVALENCE

William I. MacGregor
Bolt, Beranek, and Newman
50 Moulton St.
Cambridge, MA 02138

One of the strongest features of Pascal is the ability to define new data types. Because this ability is central to the language it is unfortunate that the original documents defining Pascal (i.e., the Jensen and Wirth "User Manual and Report" and the axiomatic definition) did not precisely state when two variables or values are of the same type, or precisely what constitutes "type checking" in an assignment statement or procedure call. Language designers have exercised their skill and imagination in attempting to resolve the ambiguities without unduly disturbing the "spirit of Pascal"; this note is one such attempt.

Recently, the BSI/ISO Working Draft of Standard Pascal was published in Pascal News #14, and this standard exhibits a particular (and carefully considered) solution to the type equivalence problem. The technique is a hybrid of name and structural equivalence; for strings and sets, the standard specifies a structural definition of type equivalence (for a discussion of name versus structural equivalence, see Welsh, Sneeringer and Hoare, "Ambiguities and insecurities in Pascal", Software Practice and Experience, N 7, 1977). While the solution is relatively direct it leaves a great deal to be desired, for instance, under the proposed interpretation all variables which are structurally integer or subrange of integer are of compatible types. Since the criterion for type equivalence is a function of the underlying structure, seemingly inconsistent cases arise. After the program fragment

```
VAR
    x   :PACKED ARRAY [1..10] OF integer;
    y   :PACKED ARRAY [1..10] OF integer;
    u   :PACKED ARRAY [1..10] OF char;
    v   :PACKED ARRAY [1..10] OF char;
```

the assignment "u:=v" is legal whereas "x:=y" is not. (The first must be permitted to include statements like "u:='abcdefghij'", and the second is presumably denied to limit the complexity of the equivalence definition and forthcoming Standard Pascal compilers.)

The rest of this note describes a different role for types and type equivalence in a Pascal-like language. The scope of the solution is strictly limited because significant extensions to the syntax of Pascal were not considered (this eliminated interesting but grandiose schemes involving a new unit of program modularity, as well as the possibility of explicit type transfer operators). The details are developed from a series of principles embodying my understanding of what strong typing means in the context of Pascal.

* * * * *

P1. Every variable has a unique type and a unique symbolic type name.

Since both the type and type name are unique, the type of a variable can be referred to by its symbolic name without ambiguity. In the interests of simplicity it seems wise to prohibit multiple names for the same type. Types are assigned to variables rather than values, because I wish to allow distinct types to exist with the same value set.

P2. All types are either predefined or created in a TYPE definition part.

The only function of the TYPE part is to define new types; the only function of the VAR part is to define new variables. As obvious as this may appear at first glance it is a very strong restriction--it implies that all types must be explicitly named in a TYPE part. For example, the Standard Pascal fragment

```
VAR
    v   :ARRAY [1..100] OF REAL;
    e   :(red,blue,green);
```

would have to be rewritten in order to conform to principle P2

```
TYPE
    vector =  ARRAY [1..100] OF REAL;
    color  =  (red,blue,green);
VAR
    v   :vector;
    e   :color;
```

This principle will force the creation of many new names in a typical program, one for each type, but at the same time it provides the basis for a simple and explicit test for type equivalence. In fact, the spread of names can be controlled in a manner described below.

P3. Every clause in a TYPE definition part (i.e., every use of the operator "=") creates a unique type.

This principle, too, seems like good common sense: the TYPE part exists to define new types. (It is interesting to note that the proposed Standard Pascal allows new types to be created in a VAR part, and doesn't require types to be created by a TYPE part!)

P4. Two variables have the same type if and only if they are declared with the same type name.

In other words we adhere to a very strict form of name equivalence. After the TYPE and VAR parts

```
TYPE
    speed = real;
    weight = real;
VAR
    a,b     :speed;
    x       :weight;
    y       :weight;
    z       :real;
```

the variables a and b have the same type (namely speed); x and y have the same type (weight) and no other type equivalences exist.

P5. In every assignment, the type of the variable on the left must be the same as the type of the expression on the right (exception: integers may be assigned to real variables).

I believe this is the simplest definition of "strong typing". To continue the previous example "a:=b" is a legal assignment but "a:=x" is not, even though the values of both a and x are real numbers. Since parameter transmission can be described in terms of assignment this principle applies to parameters in function and procedure calls; it forces an exact match between the types of formal and actual parameters, and it implies a careful interpretation of operator overloading in expressions (discussed after P7 below).

The exception is galling but historically founded. It is pervasive, as will be seen, because it implies that any type derived from integer is assignment compatible with any type derived from real.

P6. The types of all constants (simple and structured) are determined from context.

There is no way to avoid this, given P5 and the fact that variables of different types may have the same value set. Continuing the example, if the statement "a:=4.7" is legal, then by principle P5 the constant "4.7" is of type speed; but if "x:=4.7" is also legal, in this case the same value has type weight. To reconcile these cases, the type of a constant must be permitted to be a function of its context. (Note that P6 paves the way for the introduction of other types of structured constants, e.g., record and array constants; the proposed BSI type equivalence definition does not extend so easily.)

P7. A created type inherits all of the predefined operators on its underlying type, but none of the user defined functions or procedures.

This principle is admittedly a compromise. Since the ground rules forbid syntactic extensions, the promotion of operators to the new type must be automatic, and the only issue remaining is which operators should be promoted. A primal set of operators is specified in Standard Pascal; this provides a natural partitioning. (If user defined functions and procedures were promoted as well, ambiguities would result which could only be resolved through explicit typing of constants.)

An operator in the language (e.g., +) consists of a semantic action (e.g., addition) and a "signature", a template giving the types of the arguments and result of the operator (e.g., integer + integer -> integer). A user-defined type extends the set of operators available to a program, implicitly creating new operators from old ones by combining the old semantics with new signatures; each new signature is obtained from an old one by uniformly substituting the new type name for all occurrences of the base type in the old signature. For example, all programs will initially possess an operator + defined by

```
+  ==  addition;   real + real -> real
```

and in a program containing the declarations of speed and weight above the operators

```
+  ==  addition;   speed + speed -> speed
+  ==  addition;   weight + weight -> weight
```

are also available; but it would be impossible to add a "speed" to a "weight" or a "real".

With some information about context, these principles are sufficient to deduce the type of an expression or subexpression, or to select the correct operator for an overloaded operator symbol. Given

```
IF 3 < round(x/4.5 + 3.0) THEN...
```

the operators in the boolean expression must be

```
<  ==  less than;   integer < integer -> boolean
round           ;   weight -> integer
+  ==  addition ;   weight + weight -> weight
/  ==  divide   ;   weight / weight -> weight
```

and the constants 4.5 and 3.0 must both be of type weight. In a few cases involving only constants, it may not be possible to determine the constituent types, but the correct action is obvious, e.g.,

```
IF 3 IN {1,5,7,12} THEN...
```

does not permit the determination of a unique type either for the set or the base type of the set elements, but the value of the expression must be false in spite of that.

P8. A subrange is a global constraint on the set of values assumed by a variable; it does not create a new type.

Subranges are used for many different purposes; sometimes it would be useful for them to be distinct types and sometimes not. For this reason it is a good idea to accomodate both usages--if there is a simple way to do so. At this point I admit to bending the rules, and introduce one minor change to the Pascal syntax, in the form of a typed subrange. A declaration of a variable

```
i   :integer 1..10
```

means that the type of i is integer, but its values are constrained to the closed interval 1..10. A typed subrange consists of a type name followed by a subrange contained in the value set of the type. If the type name is omitted, it is assumed to be integer. If a typed subrange appears in a variable declaration, the variables have the named type; but if the typed subrange appears in the TYPE section, it participates in the creation of a (range restricted) new type, just as required by P3. For example

```
TYPE
    hour  =  1..24;
VAR
    i    :integer;
    am   :hour 1..12;
    pm   :hour 13..24;
    h    :hour;
```

The variables am, pm and h are all of type hour, and the assignments "h:=am" and "h:=pm" will always be valid; "am:=pm" will never be valid because the value sets of am and pm are disjoint; "am:=i", "pm:=i" and "h:=i" are all prohibited by type mismatch.

* * * * *

These principles lead to a view of types very different from the BSI/ISO Working Draft. It is a much more restrictive world, emphasizing type safety at the expense of flexibility. I suspect that neither approach is clearly superior for "general purpose" use, but the reader can form his own opinion.

Finally, a suggestion for controlling name proliferation appeared in an entertaining paper by Robert G. Herriot, "Towards the ideal programming language" (SIGPLAN Notices, V 12 N 3, March 1977). Herriot proposed the use of English articles ("the", "a", "an", etc.) and adjectives to create variable names. With this syntactic mechanism, the fragment

```
TYPE
    car = (ford,GM,volkswagen);
VAR
    a car                :car;
    a sports car         :car;
    a compact car        :car;
    a blue electric car  :car;
```

would declare four enumeration variables, referred to in the program text as "the car", "the sports car", "the compact car" and "the blue electric car". Thus names for variables can be directly manufactured from type names, frequently improving the program's readability.

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

FIXING PASCAL'S I/O by Richard J. Cichelli

There have been a flurry of articles advocating modifications to Pascal's file facility to improve its functionality for input/output. Here, questions regarding terminal I/O and relative record I/O will be discussed.

Many criticisms of Pascal's file facility contain arguments that Pascal's files don't support the full data set manipulation capabilities of the host's operating system. An alternate view of the situation is to ask if the problem to be solved can have its solution cleanly specified as an algorithm in Pascal. If so, request that the Pascal compiler/system writer provide an implementation complete enough to run the program efficiently. In short, buy compilers and computing systems to run your programs rather than write programs to instruct your (particular) computer.

Wirth created Pascal files. In the Revised Report Section 2, paragraph 10, Wirth defines them as sequences of components of the same type. Although an implementer may map Pascal files into sequential data sets, this isn't required by the definition. The Report doesn't seem to require that the ideas of I/O and files be associated. A valid Pascal implementation could exist on a system which lacks backing storage and a third generation file system. If this is the case for your system and you still can run your Pascal programs, what do you care? Besides, future data base oriented systems may avoid the redundancy of a "file system". The problems of named data sets and directories are obviously best dealt with in terms of local predefined (not standard) procedures.

For legible input and output (Report section 12) Pascal has a special type of file called a text file. Text files have a special substructure and special procedures and functions. Since sequences work and Pascal has appropriate facilities for manipulating them (i.e. the Pascal file primatives), it would be very strange if you couldn't make Pascal talk to terminals. Wirth specifically mentions them in the first paragraph of section 12 and, guess what, many implementors have succeeded in implementing exactly what the report calls for and having facile terminal interaction as well. One of the techniques is called "lazy I/O" and it is fully detailed in Pascal News #13.

There are those who want to put random I/O or "direct access files" into Pascal. What's Pascal missing? Surely not random access. In the Report section 2, paragraph 6, the array is discussed and specifically called a random access structure. "But", you say, "I can't fit big direct access files in core". Every implementation of Pascal is likely to have some restrictions. Perhaps an array will need to be stored on bulk storage. Would you embed this limitation in the language and in your algorithms and programs? If you need to worry about a hierarchy of memory access facilities in these days of virtual memory, etc, then a pragma or compiler directive might be the appropriate mechanism for suggesting to a particular compiler that certain data be placed on backing store. Note: There is no prohibition to passing arrays (e.g. an implementation relative records I/O) as program parameters. See the Report section 13. Program parameters can reference any external object. It is only suggested that these are "(usually files)". Thus arrays and pointer based data structures can be external objects to Pascal programs. (The "(usually files)" reference has been removed from the current draft standard document.)

Although doing relative record I/O with Pascal arrays may seem strange at first, adding the unnecessary notion of memory hierarchies to the language is far worse. The IBM System/38 has a uniform 48 bit addressing mechanism. A System/38 applications programmer does quite well while being unaware of the storage location of his data whether it be cache, core, disk buffer or on disk. If the 38 can be said to auger the future, then certainly Pascal shouldn't take a step backwards and introduce concepts which provide no additional functionality

In summary, fixing Pascal's I/O only requires implementing what the Report suggests.

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

JAREK DEMINET, M.Sc.
JOANNA WISNIEWSKA
Institute of Informatics
University of Warsaw
P.O.Box 1210
00-901 Warszawa

POLAND

Simpascal

Introduction

This article presents a new extension (called Simpascal) to Pascal. The goal of this extension was to provide facilities for simulating discrete time systems in the way similar to the one adopted in Simula. This goal has been achieved with no changes in the original Pascal compilers, but rather by use of some run-time routines. Simpascal has been implemented on CDC CYBER73 and IBM 360.

Background

Simpascal was designed as a part of the OSKit Project (simulation of operating systems [1]) at the Institute of Informatics, University of Warsaw. Those extensions were necessary since existing standard Pascal facilities didn't allow one to write a simulator in this language. The reason for creating a new tool instead of using Simula was mainly better performance of the Pascal object code. Besides, all other parts of the project (data input preparation and output analysis) had been already written in Pascal.

A general design of Simpascal and its implementation on the CYBER73 were made by Jarek Deminet, while some improvements and the 360 version were prepared by Joanna Wisniewska. A standard Pascal compiler was used on the CYBER73. 360 compiler was produced at the Institute of Computer Science, Polish Academy of Science. The whole work lasted approximately 6 weeks.

Description

A simulator in Pascal (as in Simula) consists of some number of coroutines, each of which implements one process. At any given time one of them is *active* and the others are *suspended*. Some of the latter may be *ready* to run and wait in a so-called *Sequential Set* (SQS), other are *blocked*. SQS is ordered according to increasing *time*, which is an attribute of each process. Full description of this idea may be found in [2]. From this point on, a term *routine* will mean either a coroutine, subroutine or the main program, while a *subroutine* will be either a procedure or a function.

In order to provide all expected functions the following subroutines were implemented:

`function Create (procedure P):coroutineID;`
Creates a new process (coroutine), with the same attributes and the body as in the procedure given as a parameter. This coroutine is started; after an initial part it should call `Detach` (see below). Control then returns to a creator, and the function returns as its value a unique coroutine identifier. The first routine calling this function is called a *root* of the whole set of coroutines. There is a restriction that `Create` may be later called either by the root or by any other coroutine, but not in its initial part (*i.e.* no nesting of `Create` calls is allowed).

`procedure Detach;`
Finishes an initial part of a coroutine and returns control to its creator.

`procedure Start (C:coroutineID; maxtime:real);`
Starts the coroutine `C`, thus initiating the whole simulation. It should be pointed that, unlike in Simula, an root is not a coroutine itself and may be resumed only after finishing the simulation. Simulation ends as soon as there is no process in the SQS with its time less then the `maxtime` parameter of `Start`.
This routine may be called only by the root.

`procedure Activate (C:coroutineID; delay:real);`
Makes the coroutine `C` ready, *i.e.* inserts it into the SQS. Its time will be equal to the time of the currently active (current) coroutine increased by `delay`. If `delay` is negative, then the coroutine `C` will be resumed immediately (becoming active), and the current coroutine will be suspended.

`procedure Pass (C:coroutineID);`
Acts similarily to `Activate (C,-1)` , but also removes the current coroutine from the SQS.

`procedure Cancel (C:coroutineID);`
Removes the coroutine `C` from the SQS. If that was the current coroutine, the next coroutine from the SQS is resumed.

`function Time (C:coroutineID):real;`
Returns time of the coroutine `C`.

`procedure Hold (increment:real);`
Suspends the current coroutine, increases its time by `increment` and resumes the first coroutine from the SQS.

`function This:coroutineID;`
Returns an ID of the current coroutine.

There is one very unpleasant and artificial restriction for a call of the so-called *special routines*, which may change an active coroutine (*i.e.* `Activate`, `Pass`, `Cancel` and `Hold`). If any of those subroutines is called from a Simpascal subroutine, called in turn (directly or indirectly) from a coroutine, then all subroutines down to the level of the coroutine will be immediately terminated. That means that the coroutine is suspended and reactivated always at its own level. This concept was called *husking* and is necessary to ensure stack consistency.

Implementation

The data structure on which Simpascal subroutines operate is very similar in both implementations so it will be presented here in a relatively machine-independent form.

Each instantiation of every Pascal routine is defined by a segment on the stack (the routine is called an *owner* of this segment). Each such segment (except the first one, corresponding to the main program) consists generally of two parts:

Environment definition
: Contains all information necessary to refer non-local objects, to safely execute a return jump, and to perform an error handling action if necessary.

Local data
: Contains local variables (which include also parameters of the call, compiler-generated auxiliary variables and space for registers saved in case of further routine calls). Generally, this part is of no interest for Simpascal, except for register saving space.

A base (an address of the first word) of the segment of the current routine is pointed by one of the registers (a B register on CDC, a general purpose register on IBM), which will be called a Base Register (BReg). Also the first free location above a top of the stack is pointed by a register (Top Register or TReg).

In case of ordinary Pascal subroutines information in the environment definition is as follows:

Static Link (SLink)
: Points to a base of the segment defining the latest instantiation of the routine in which the segment's owner was declared. A chain of those links defines an access path to all non-local objects.
This link is created always by the caller, according to its own access path.

Dynamic Link (DLink)
: Points to a base of the previous segment on the stack, *i.e.* the segment corresponding to the routine which called the owner of this segment. It is used to restore the BReg before return and to produce a Post-Mortem Dump should an error arrive.
This link is created by the routine itself using the old value of BReg.

Return Address (RAddr)
: Contains the address to which control shuld be transferred in a return jump. This address is provided by a caller (passed through a register).

Figure 1 presents the general structure of the Pascal stack.

The same data structure had to be adopted in Simpascal, since the code of coroutines was to be the same as for normal subroutines. Several assumptions had to be made, however, to ensure a consistency of the structure:

All coroutine segments occupy a contiguous space on the stack, directly above the segment defining the root of the system (there may be no other segments in between).

The stack of only one coroutine at any particular time (the active, or current coroutine) may consist of more than one segment. This would mean that no action which implies a change of the active coroutine may be undertaken from any level other than the level of the coroutine itself. To allow creating of user-defined control transfer subroutines the concept of *husking* (described above) was adopted. Its implementation is very simple: any special subroutine removes from the stack all segments from above the block of the coroutine segments.

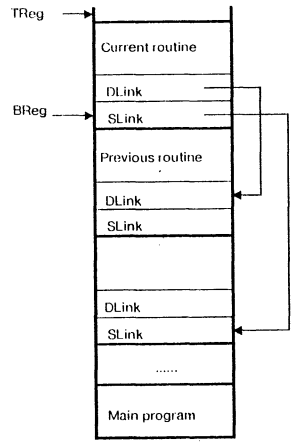


Figure 1. Pascal stack structure.

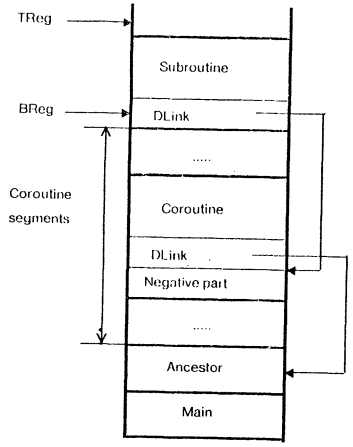


Figure 2. Simpascal stack (only some parts shown)

The segment for each coroutine was changed in a manner invisible to ordinary routine code. First, a *negative part* was added. It contains a restart address for an inactive coroutine, and also some additional information (time, status and some pointers) used by routines which handle and sequence processes. The meaning of some standard fields was also modified. Since the assumption is that a coroutine will never execute a return jump (because it would destroy the stack structure), RAddr points to an error-handling routine.

DLink, in turn, no longer points to the previous segment on the stack, since it is not intended to be used to update the BReg. Because of some functions played by DLink during standard error handling, it was decided that this it should point to a base of the root's segment.

Contents of BReg, TReg and SLink were left unchanged.

Figure 2 illustrates the general stack structure in Simpascal.

In order to have such a structure, the following actions have to be performed by `Create` before calling a coroutine:

- setting BReg to a base of the root;
- incrementing TReg by the size of the negative part of the segment;
- setting SLing according to information which is always a part of the actual-parameter-descriptor in a call-by-procedure in Pascal.

Results

Several programs have already been written in Simpascal and run on both machines, fulfilling all expectations. A comparison with Simula shows that a program in Simpascal needs 50 to 80% less memory and 50 to 70% less time. This is mainly due to much simpler memory structure allowing better performance of the code.

References

[1] Leppert M., Madey J., Schroff R. : *ITS Status Report*
Report 7739, Technisches Universitat Munchen, Munich, Germany;
Report 63, Instytut Informatyki Uniwersytetu Warszawskiego, Warsaw, Poland

[2] *Simula 67 Common Base Language*
Publ. no. S-22, Norwegian Computing Center, Oslo, Norway

☆ ☆ ☆ ☆ ☆ ☆ ☆ ☆ ☆

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

The University of Tasmania

Postal Address: Box 252C, G.P.O., Hobart, Tasmania, Australia 7001

Telephone: 23 0561. Cables 'Tasuni' Telex: 58150 UNTAS

IN REPLY PLEASE QUOTE:

FILE NO. .. ....

IF TELEPHONING OR CALLING

ASK FOR .. ..

Some observations on Pascal and personal style

Arthur Sale

Tasmania, 1979 June

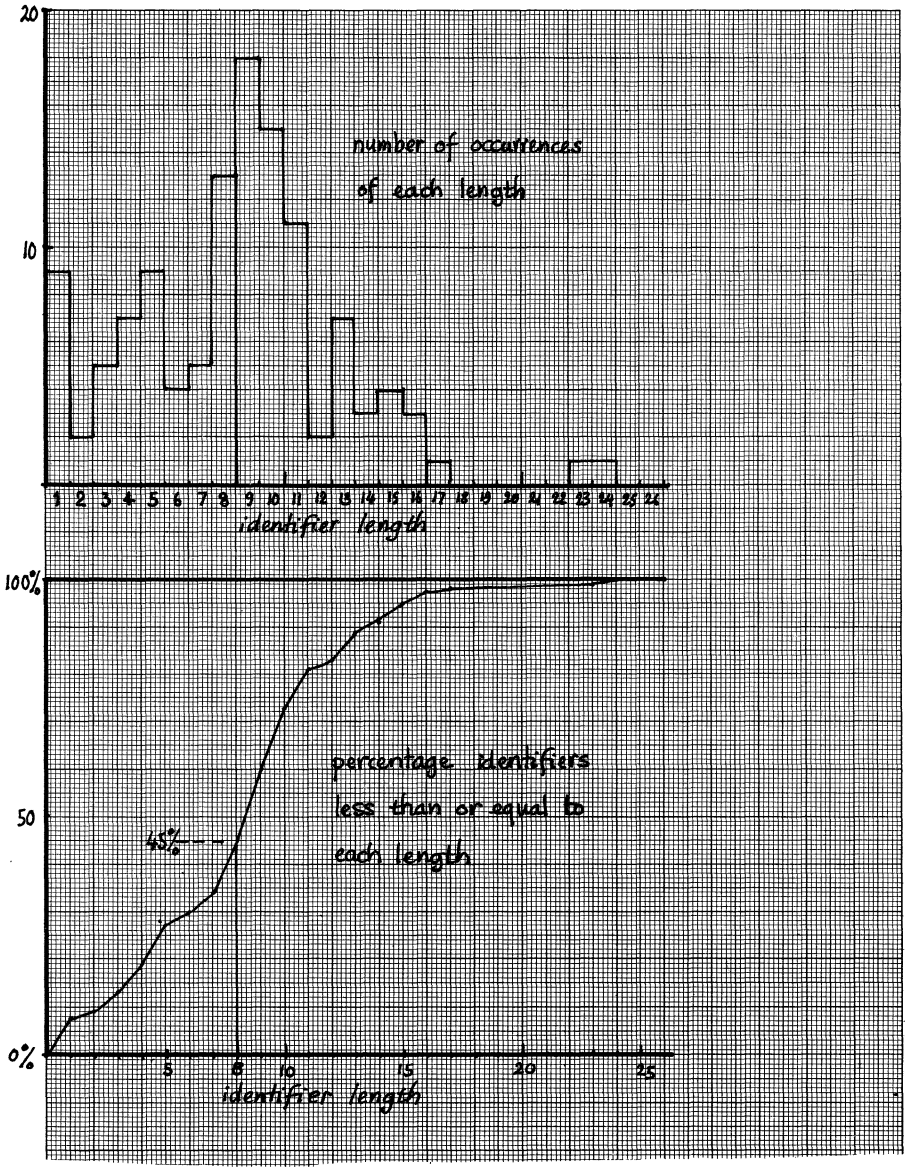
Background

Recently, arising out of a course I gave for microprocessor engineers and their possible use of Pascal, I had to write a program of around 800 lines to control a hot-plate assembly (as might be installed in a home with provision for switching the hot-plates individually up or down at selected times). The purpose of the program was to demonstrate the viability (and superiority!) of Pascal for microprocessor purposes over assembly code or Fortran. The experiment was a demonstrable success, taking one man-day to write together with its correctness proof, and another man-day to transform the abstract program into one having some useful properties for micro-processor Pascal compilers and run-time support. The experiment will be reported elsewhere; by contrast the writing of the consequent paper has consumed over a man-week, and nearer two...

However, in the course of writing this up, I came across some interesting facts I should like to share with the readers of Pascal News. They relate to personal stylistics, and use of Pascal's features. None of the reported statistics here were considered specifically while writing the program: they reflect a personal style.

Identifiers

The program contains 120 identifiers, and one label (as a consequence of a transformation to eliminate a task). The length distribution of the identifiers is shown in Figure 1.



It is interesting to note that approximately 55% exceed 8 characters in length, and approximately 27½% exceed 10 characters in length. These correspond to the significance limits of the Pascal Standard and the Control Data Cyber compiler. The Burroughs B6700 compiler I used has, of course, no limit on significance.

Since the B6700 compiler is good in this respect, it is possible to write programs which work on the B6700, but which give rise to compiler error messages (or worse, altered and undetected scope renaming) on systems with limited significance. How often does this occur? Fortunately, the STANDARD option on the B6700 compiler checks the possibility of any such events. The answer seems to be: surprizingly often. In previous programs I have seldom been able to escape changing an identifier name to avoid problems elsewhere, and it happened twice in this program. The instances were:

```
numberofevents    {an integer variable}
NumberOfPlates    {a constant, altered to NoOfPlates}

DisplayType       {the type of the display register}
DisplayTime       {a procedure, altered to DisplayATime}
```

I draw the conclusion that any compiler that has a significance limit greater than 8 characters ought to perform the same checks; software I receive from elsewhere often exhibits the same problem. I also conclude that the 8-character limit is a mistake, and should never have been introduced into Pascal.

The B6700 compiler also produces as a by-product of this checking a list of instances of renaming under the scope rules. None were reported in this program at all, which surprized me. Usually *i* and *j* crop up with monotonous regularity, but in this case it appeared that the lesser numeric orientation and the program structure minimized this.

Letter Cases

As the examples above indicate, the compiler accepts either letter case in accordance with the Pascal Standard, and I write programs in predominantly lower-case letters. I dislike the practice of capitalizing the reserved words as it has a bad effect on readability for me. However, during the course of this program I found myself falling into a practice which I had never used before, but which seemed to be useful. I offer it as an example of the differences in personal style that can arise with a little thought devoted to stylistics.

The practice I adopted, more or less by chance at first, was to write variables in all-lower-case, as in *numberofevents*, but constants, types, and procedures in mixed-cases, as in *NoOfPlates* or *DisplayATime*. Rationalizing it after the event, I noted that variables often have shorter and less complex names than other objects and thus may have less need of extra lexical cues, and procedure names are often the longest and most complex. Sometimes these are a verb-phrase, while variable names are more noun-like.

The practice improved my understanding of the program, mainly because I could detect in expressions which were variables and which constants. Such slight cues are worth a lot more to me than emphasizing reserved words (which I know very well). Example:

```
if (time = LastMinuteOfDay) then begin
```

I am not yet sure whether this will be a stable feature of my future style.

Line Layout

I used my usual line layout and indentation rules, reported in Sale [1978], and had no need to edit or correct any semicolons or ends. A consistent style minimizes these trivial but annoying errors.

Comments

I classified the comments into three categories:

(a) Marker comments, used to assist picking out corresponding points in a program, typically attached to an end to show what it is the end of, or to pick out a procedure name by underlining. Little semantic content.

(b) Procedure heading comments. These have considerable semantic content, and outline the purpose of the procedure.

(c) In-text comments, which either give additional information relating to the execution, or explain definitional points. They vary all the way from a hint:

```
{Midnight changeover}
```

to an assertion:

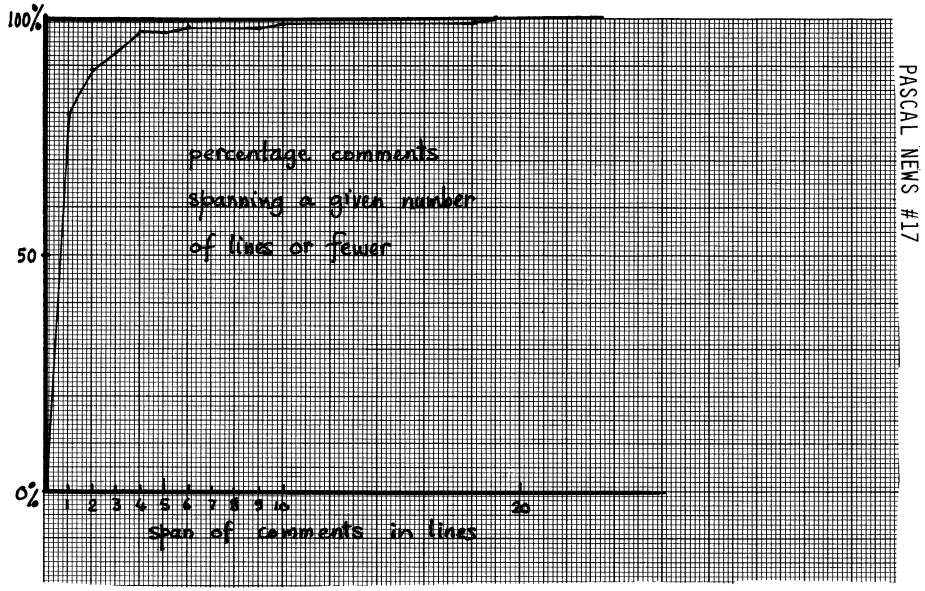
```
{Re-establishing the invariant:
    Ri = "All events up to and including the one pointed
         to by the 'preceding' pointer are due to occur
         before or simultaneously with the new one.  Also
         if state=Exit there are no more records that
         satisfy this criterion."
```

The comment characteristics are shown below.

Kind of comment	no	lines spanned	% lines spanned
Marker	36	36	19%
Header	18	67	35%
In-text	67	87	46%
TOTAL	121	190	100%

The closing comment marker ("}") was always the last non-blank character of the line it appeared on. Since one-line comments make up 80% of the total number of comments, and 51% of the total number of lines spanned, here is support for the idea that comments delimited by end-of-line require no more keystrokes than bracketted comments. (Apart from other, better, reasons for preferring them.)

The distribution of comment lengths, shown in Figure 2, emphasizes this. It is certainly influenced by my habit of putting correctness assertions and hints in the code body, thus reducing the size of procedure header comments. (The comments often share lines with code, so do not make the mistake of assuming that the program contains 190 lines of waffle together with the 157 blank layout lines).



Procedures and Functions

Having arrived at a suitable transformation level by eliminating tasks from the conceptual solution and substituting interrupt-driven procedures (Yes, I know they aren't standard), the resulting program had 18 procedures/functions, including the main program. Other statistics are:

Procedures	Functions	Program
15 (83%)	2 (11%)	1 (6%)

Parameters:	0	1	2
	13 (76%)	3 (18%)	1 (6%)

The low frequency of parameters is explained by the nature of several of the procedures: they are refinements. In fact six of the parameterless procedures are called from only one place each, and a microprocessor engineer might well apply a transform to put their code in-line and their local data in the caller's stack. Personally, I exert pressure on compiler suppliers to make their compilers do it automatically: detecting the once-only call is not difficult for a multi-pass compiler. On the B6700 such a transformation would save 54 bytes of code out of a total of 2304 (2.3%), and would also speed up the execution slightly.

The maximum level of procedure nesting is three, and this occurs 7 times. This is astonishingly low for me, since my refinements often creep up into the 10 to 12 levels deep. Analysing it after the event, I conclude that the low nesting level here is due (a) to the complexity of this problem being in task interlocking, not in algorithm complexity, and (b) to several refinements being pushed to outer levels for use in several contexts (by the sub-tasks).

Types

As might be expected, real numbers are not needed in this problem. The usage of different types in the program is shown below:

	definitions	uses in var or type
boolean	(1)	2
integer	(1)	0
char	(1)	0
real	(1)	0
user-defined scalars	6	7
subranges of scalars	1	1
subranges of integer	9	30
records	1	1
arrays	3	8
sets	4	8
pointer types	1	4
files	0	0

The absence of integers arises naturally because no negative numbers occur in this problem, and because the range of every integral value is predictable. Only innate laziness allowed one of my favourite types:

```
Natural = 0 .. Maxint;
```

in to substitute for the type of a value parameter which ought to have had a special type declared for it in the outermost block:

```
TwoDaysWorthOfMinutes = 0 .. 2879;   {2*24*60 - 1}
```

I salved my conscience by adding a comment to this effect, which probably took more time doing it right...

Of some interest is the ratio of user-defined scalars to uses of predefined types (7 : 2). This is a measure which I take as roughly indicative of a switch from other language thinking to Pascal (or abstract) thinking.

The problem isn't big enough to draw any more conclusions.

Boolean expressions

Some people, on seeing my programs, adopt a knowing look and say, "You used to be a Fortran programmer, weren't you?" and point to an example like:

```
if (eventlist = 0) then begin
```

Since this is total misunderstanding, it deserves a few words. I usually put parentheses around every relational expression I write. The prime reason is that I find it greatly improves the readability of the program in that the limits of some complex expression can be more readily found, as for example in:

```
if (modulocounter in pattern[plate[i]]) then begin
```

But having done this for a long time, it confers several other benefits:

(a) I almost never make mistakes in writing expressions which the Pascal syntax will parse in a way I didn't intend. (The few priority levels are well-known as a trap).
(b) I have to devote less thought to trivia while writing programs, and therefore more thought to correctness proofs, simply because I use codified rules.

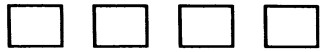
To illustrate the point, the same thing happens in the following example:

```
IsT1BeforeT2 := (t1 < t2);
```

Summary

The purpose of this little letter is to give you some insight into some personal stylistics in the hope that you will examine your own equally carefully and ask yourself whence they came and why. Pascal is no language for nongs who mindlessly copy others. I also hope it may give some ideas to compiler-suppliers on the sorts of things I do. If you ever want to please me, here are some hints. Preserve the abstractions and make any limits on what I can do at what I call *virtual infinity*....

Arthur Sale



Open Forum for Members

Yale University *New Haven, Connecticut 06510*

SCHOOL OF MEDICINE
333 Cedar Street
Section of Laboratory Medicine

January 23, 1980

Andy Mickel
University Computer Center
University of Minnesota
Minneapolis, Minnesota 55455

Dear Andy:

Yesterday I called and spoke to Rick Marcus about a bug I have found in ID2ID. My version is attached, together with the data that showed the fault, and the symbol table progression in the original system.

Please consider this as a letter to PUG, and pass it on accordingly. I attach a second copy for the purpose.

On Pascal Standards, I have several observations. First, based on Bob Fraley's HP3000 Pascal Compiler, I feel the need for a standard procedure

PROMPT (FILE)

which will have the effect of a writteln without causing a line-feed or carriage-return. This is required for interactive use, where the underlying system buffers output. The procedure will flush the buffers. Wherever the I/O system is direct, the procedure is not needed and need not generate any code.

I firmly approve of the "otherwise" clause in the case statement, and also feel it should be extended to variant records. I.E.

```
      A        =    Record
   Case B      :    Type of (
      C        :    Ctype )
      D        :    Dtype )
  Otherwise    :    Elsetype ))
```

DISPOSE is often replaced by Mark/Release, which should be an available option in the standard. DISPOSE must always require a garbage collector, and thus a good deal of run-time. However, systems not implementing dispose should generate a null procedure for source compatibility, and similarly for Mark/Release. Note that implementation of Mark/Release on systems that provide (new) storage to various processes from a common pool must implement the equivalent of dispose for a release. However,

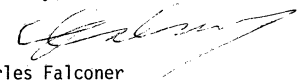
these systems are complex by definition, and thus a full DISPOSE is probably not excessive. In this case release effectively signals the garbage disposal system to function.

An extension sorely needed is simple arithmetic in constant definitions, allowing all compile time constants to be slaved to a single definition. Similarly the use of ORD and CHR functions in constant definitions would be useful.

Implementation of goto's out of procedures is virtually impossible (at reasonable cost) on many machines. The HP3000 is an example. I would therefore recommend that the standard does not require these, and that they be considered an extension. Logically, I have never found such goto's necessary, and in addition such use customizes code segments to any overall program, preventing direct re-use.

I am also running the USCD Pascal System, VER II.0. Users should be warned that, as supplied, this does not detect integer overflows, (at least on 8080/Z80 Systems), and that the complement of -32768 is 0!! with no warning.Some stack overflows can occur without trapping in addition. My revised interpreter cures these problems, when many system programs proceed to crash on integer overflow, and thus the overflow check has been made switchable. The USCD System does not detect EOF on the remote files, and thus cannot read text files remotely without considerable contortions.

Sincerely,



Charles Falconer
Chief Instrument Engineer

CF:tmm

Enclosures

Carnegie-Mellon University
Department of Computer Science
Schenley Park
Pittsburgh, Pennsylvania 15213

January 29, 1980

A.M. Addyman
Department of Computer Science
University of Manchester
Oxford Road
Manchester M13 9PL
England

Dear Professor Addyman:

I was delighted to see the proposed Pascal standard in Pascal News. In general, I think the proposal is excellent. However, there were a few points that troubled me.

- Textfiles. 6.4.24 seems to require that every textfile end with a linemarker. Is that intentional? If so, must closing a file(used for writing) force a linemarker to be output if one does not already end the file?

- Pages. It seems bizarre to include a standard Page procedure without specifying the effect on the file or including a procedure to test for end of page . I propose making the procedure optional, but if it is included, require that a page marker be written which is (like a linemarker) read as a blank, and that an Eop (end-of-page) predicate be included as well. Additional questions: Should Eop imply Eoln? Should Page force a Writeln automatically?

- The CASE statement. I must say I am surprised the OTHERS clause was not included in the standard. I'm equally unhappy (but less surprised) that subranges were not to be permitted in the case-constant list.

- Numeric output. 6.9.3 requires a leading blank for a number that fits in the output field, while no leading blank is required if it does not. So, in the case of a number whose width is the same as the fieldwidth, the number is printed out in just that fieldwidth without a leading blank. I suggest rewriting the specification so that this is clear - by noting that Ø rather than 1 leading blank is required.

I have seen the notation Write(Val: 1) used to mean: Use the smallest possible fieldwidth. A cute use of the specifications, but its obscurity is not in the spirit of the language. Perhaps Write(Val) ought to print Val in the smallest fieldwidth possible (no leading blanks either!) while a fixed fieldwidth would be used only if specified. This would unquestionably be the most pleasant solution for most users, especially novices.

The Write(Val: 1) idiom is deficient for another reason. Many implementors have chosen to implement output in an undersized field by writing out asterisks. A good case can be made for this, and I suspect many Pascal implementors will continue to do so despite the standard.

Sincerely,

Ellis Cohen

BRITISH COLUMBIA HYDRO AND POWER AUTHORITY

Red Stripe Computer Trailer
Gas Division
3777 Lougheed Highway
Burnaby, B.C.
V5C 3Y3 CANADA

1980 January 22

Dear Pug

I wrote a while ago about banning the marriage of Pascal and EBCDIC.
I think I stated a decent character set should have the following property
"ORD('9')-ORD('0') should be 8" That should read "ORD('9')-ORD('0') should be 9".

If you decide the publish that letter, please correct the mistake.
Please do not publish this letter.

Thanks

WINTHROP PUBLISHERS, INC., 17 dunster st., cambridge, mass. 02138 tel: 617-868-1750

January 8, 1980

Professor Andy Mickel
University Computer Center
227 Experimental Engineering Bldg
208 SE Union Street
University of Minnesota
Minneapolis, MN 55455

Dear Andy,

I'm a little concerned about some possible unintended effects of your brief book reviews section on page 8 of Pascal News, No. 15.

You quoted a table from a review by Jan Hext, of the University of Sydney, comparing Pascal textbooks in their coverage of the language. I am concerned that, taken out of context, that table may scare potential readers away from our book by Conway, Gries, and Zimmerman, A PRIMER ON PASCAL, the second edition of which is due this spring.

There is no question that the coverage of Pascal in that book is not nearly as extensive as many other books (although in the new edition it will be somewhat more so), but taken out of context, it looks like you are rating the book in general as "poor." The reviews in my files indicate, of course, that the book is arguably the best introduction to programming using Pascal as a vehicle, and for such a use might well be much more appropriate than a book which is a more thorough rendering of the language but less helpful in learning to program. So, while I do not quarrel for a moment with Professor Hext's analysis of what this book is not, I wish to rush to the barricades to reaffirm what, on the other hand, it is.

Thanks for listening.

Best regards,
Chuck mw
Charles F. Durang
Editor, Computer Science

CFD/mw

THE CITY COLLEGE
OF
THE CITY UNIVERSITY OF NEW YORK
NEW YORK, N.Y. 10031

SOPHIE DAVIS SCHOOL OF BIOMEDICAL EDUCATION

(212) 690-6629, 8255

Wednesday, January 30th, 1980

Rick Shaw
PASCAL User's Group
Digital Equipment Corporation
5775 Peachtree Dunwoody Road
Atlanta, Georgia 30342

Dear Rick;

Enclosed is my personal check for $26.00; please enter my subscription/membership to PASCAL News for this academic year 1979/80, and also send the previous two years' back issues 9 - 16. I would be glad to pay Xeroxing and mailing expenses (within reason) if somebody could furnish copies of your extinct issues 1 - 8.

In our mammoth CUNY University Computer Center (Amdahl 470/V6 and IBM 3033 under OS/MVT and ASP; IBM 3031 under VM and CMS), Stony Brook PASCAL 1.3 is standard, and Version 2S was just added to a test library last week. (I gather from the documentation that both are rather limited in complex applications - for example, no external files...) Although I know of no campus among our 20 where PASCAL is the prime teaching language, faculty and student use is clearly on the rise; we've just brought up 2 PASCALs on a PDP-10 here in the CCNY Science Building.

I am involved in bringing up an orphan Z-80 microcomputer from the defunct Digital Group in Denver; besides opscan test grading, the primary application will be bibliographic citation retrieval from a hybrid collection of about 8,000 articles. I am presently working up the necessary software package for this operation in PASCAL, using bit-string inverted lists hung from a B-tree.

With the possibility of a brief trip to Switzerland this April, I have considered arranging a visit with Professor Wirth: if anybody else has done so, expecially recently, I'd love to hear from him as soon as possible. PASCAL was my native language at SUNY Stony Brook, and I'm very thankful for that. I'm eager to meet other New York City PASCAL users.

Sincerely yours;

Alan N. Bloch, M.P.H.
CCNY Biomed J 910 C1

encl.

KERN INSTRUMENTS, INC.
GENEVA ROAD • BREWSTER, NEW YORK 10509

Kern SWISS

TELEPHONE:
(914) 279-5095
TELEX:
969624

January 15, 1980

Mr. Rick Shaw
Digital Equipment Corporation
Pascal User's Group
5775 Peachtree Dunwoody Road
Atlanta, GA 30342

Dear Rick:

While renewing my subscription, I am taking the opportunity to say a few words.

I have used two Pascal systems in my work here; initially, a Northwest Microcomputer 85/P with UCSD Pascal, and now a PDP-11 with RT-11 operating system and OMSI, Pascal I version 1.1. Both have advantages (and disadvantages). The UCSD operating system (with CP/M utilities) was fantastic, especially the editor. However, I/O handling (I wanted interrupts) was poor. With RT-11, I can use all the I/O facilities of this excellent operating system, but OMSI doesn't support them very well. Hopefully, this will be fixed in version 2 which is due any day now. I'm also disappointed that several Pascal features I used quite heavily with the UCSD system are not implemented in OMSI Pascal I, particularly the Pack and Unpack functions. These are very convenient for formatting and unformatting I/O records used in certain peripherals.

I see almost weekly announcements concerning new Pascal compilers and machines. Now that most of the established computer manufacturers have taken up the cause, we can say that Pascal has arrived. So much so in fact, that I would not have resubscribed to PUG if not for Arthur Sale's recent issue describing the Validation Suite. Congratulations to Prof. Sale and his group.

Now it's up to us Pascalers to encourage the compiler writers to meet the standard and implement any extensions in an acceptable manner.

Good luck, Rick!

Sincerely yours,

KERN INSTRUMENTS, INC.

T. P. Roberts
Photogrammetric Systems Engineer

TPR:pm

BRITISH COLUMBIA HYDRO AND POWER AUTHORITY

970 BURRARD STREET
VANCOUVER, B.C.
V6Z 1Y3
TELEX 04-54395

1979 December 31

Dear PUG

re: outlawing EBCDIC and Pascal marriage

I have tried to write some text tidying routines with the University of B.C. Pascal compiler under MTS. It uses EBCDIC as its underlying code. Arrgh!

ORD('Z')-ORD('A') should be 25 in all decent Pascal implementations.
ORD('z')-ORD('a') should also be 25. There should exist a magic number m such that you can do lower to upper case conversions.
ORD('9')-ORD('0') should be 8. (Even EBCDIC gets that right!)
ORD(' ') should be less than ORD('A'), ORD('a'), and ORD('0').

ASCII has these properties. EBCDIC does not. It is thus difficult to write portable code.

I suggest that any Pascal standard insist that an "excellent" rated compiler provide a compile-time switch to insist that all internal character codes be ASCII even if this means translation in and out. Alternatively, Pascals that live in an EBCDIC environment that wish to manipulate all 256 characters should work internally on a modified EBCDIC that has the above nice properties. A compiler that could not provide this option could only obtain a "reasonable" rating.

To indicate the horrors of EBCDIC, consider that none of the following code works as you would expect.

```
if C in ['a'..'z'] then S1;      if C >= 'a' and C <= 'z' then S3;

for C := 'A' to 'Z' do S2;
```

It is also impossible to write (I hope I am wrong) decent hashing algorithms and random number generators that are truly portable (ie. give the same answers in all implementations). Perhaps "excellent" rated compilers should also provide some extra builtin functions for these tasks. It wouldn't hurt to define their names and parameters now.

Roedy Green

Roedy Green

COMPUTER SCIENCE PRESS INC.

9125 FALL RIVER LANE
POTOMAC, MD. 20854
[301] — 299-2040

November 27, 1979

Dr. Andy Mickel
Editor
Pascal Newsletter
University of Minnesota
University Computer Center
227 Experimental Engineering Building
Minneapolis, Minnesota 55455

Dear Dr. Mickel:

In the 15th issue of Pascal News on page 8 you inadvertently omitted the list for our book PASCAL: An Introduction to Methodical Programming by William Findlay and David Watt in an article comparing the available Pascal books. It was listed in the table at the bottom of the page. We would appreciate it if you would correct this ommission. Pascal: An Introduction to Methodical Programming is published for the United States and Canada by Computer Science Press, Inc. @ $11.95. it is available and published throughout the remainder of the world by Pitman Publishing Ltd., 39 Parker Street, London, England WC2B 5PB.

In its first year Computer Science Press has sold over 12,000 copies of Pascal: An Introduction to Methodical Programming within the United States and Canada. We also believe that a much more meaningful comparison and evaluation of books can be obtained by the basis of the universities and colleges which are using it. Our book has been adopted at over 50 schools within the United States and Canada including:

Arcadia University
Albright College
Brock University
Broome Community College
California State University at Long Beach
Cariboo College
Case Western Reserve University
College of William and Mary
Dalhousie University
Dickinson College
Fairleigh Dickinson University
Framingham State University
Iowa State University at Ames
John Brown University
Kansas Wesleyan University
E. R. Lauren University
LeTourneau College
Loyola University
Lucas College
Marian College
Marquette University
Merritt College
Montana State University at Bozeman
Moravian College
Morningside College
North Carolina State University at Raleigh
Northampton Community College
Northeastern University
Plymouth State College
Purdue University at West Lafayette
Rollins College
Rosemont College
Sonoma State College
Southern Methodist University
Stanford University
Temple University

COMPUTER SCIENCE PRESS INC.

9125 FALL RIVER LANE
POTOMAC, MD. 20854
[301] — 299-2040

Texas Technological University
Thames Valley State Technical Coll.
Towson State University
Union College
U.S. Military Academy at West Point
U. of California at Berkeley
U. of Chicago
U. of Houston at Clear Lake City
U. of Maryland at College Park
U. of Massachusetts at Amherst
U. of Mississippi at University
U. of Oregon at Eugene
U. of the Pacific at Stockton
U. of Pennsylvania at Philadelphia
U.of Saskatchewan
U. of South Carolina at Columbia
U. of Southern California at Los Angeles
U. of Texas at Austin
U. of Utah at Salt Lake City
U. of Washington at Seattle
Villanova University

WE INVITE ALL COLLEGE PROFESSORS WITHIN THE UNITED STATES AND CANADA TO WRITE TO COMPUTER SCIENCE PRESS AND REQUEST A COMPLIMENTARY COPY OF PASCAL: AN INTRODUCTION TO METHODICAL PROGRAMMING. Please write on school stationary, identifying the current text, the course name and number, as well as the anticipated annual enrollment, and we will be happy to let you determine for yourselves which is the best book for teaching Pascal.

We would also like to call to your attention our **short** course program offered through our Computer Science Education Extension Division which will offer 2-3 day courses on Pascal on:

March 24-28 at San Francisco
May 18-19 at Anaheim (preceding the NCC)

Thank you.

Sincerely yours,

Barbara B Friedman

Barbara B. Friedman
President

BF:cw

PRINDLE AND PATRICK ARCHITECTS: PLANNERS

"MEMBERS: THE AMERICAN INSTITUTE OF ARCHITECTS"

October 17, 1979

Pascal User's Group
c/o Andrew Mickel,
University of Minnesota,
227 Experimental Engineering,
Minneapolis, MN 55455

Re: Pascal User's Group and Pascal Newsletter

Dear Sir:

I would like to get information on the Pascal User's Group, especially, as soon as possible to get the Pascal Newsletter, including back issues if possible. I would like to join the organization and find out all I can as I am contemplating committing this system to extensive use of Pascal, although I am not at present a Pascal user. I would like to learn something about the availability of Pascal software, either to swap or sell.

The company I work for is an architecture firm which has a PDP-11/34 running RT-11 and TSX (TSX provides several virtual RT-11 single job monitors with some limitations and some additions, if you are familiar with DEC's RT-11.). Our applications are Accounting, Word Processing, and some statistics and simulation. We hope someday to get into some graphics. Right now there is an awful lot of awful assembler stuff around here which must one way or another be transformed into something more portable.

Another bit of background is that I am one of the first users of the Whitesmith's Ltd. C compiler, which satisfies the specification given in Kernighan and Ritchie's book with one addition, which is that different "typedef"s can have elements with the same name. I.e. there can be an A.x and a B.x. According to Kernighan and Ritchie, this is not allowed in regular C, which is very peculiar (It can be disabled in Whitesmith's C for compatibility). Whitesmith's also says they have been using their own UNIX-compatible O.S. (will run UNIX binary programs) for about a year now, and will soon be selling it for much less than the cost of UNIX. I estimate that one (such as a very serious hobbiest) could have a quasi-UNIX system within a year for under $10,000. The rub is that Bell Labs is currently trying to make it appear as if UNIX user's society software cannot be spread around to non-members, at least that is the impression I get. But a number of sources from DEC to Yourdan, Inc., to Whitesmith's tell me they don't have a legal leg to stand on. But who will get the ball rolling?

I don't know if all this interests you or not, but I thought there was a fair chance it might, and that you might be able to lead me to some help in finding or helping to establish services that would do for C what you are doing for Pascal. My own impression is that C and Pascal are quite complementary, C being a better systems language, and Pascal being better for many, or even most applications.

Sincerely,

Hal Morris

Hal Morris
System Manager

BRITISH COLUMBIA HYDRO AND POWER AUTHORITY

Red Stripe Computer Trailer
Gas Division
3777 Lougheed Highway
Burnaby, B.C.
CANADA V5C 3Y3 (604)298-1311 loc 372

1979 November 20

Dear Pascal Standardizers:

The key beauty to Pascal is that if you take a valid Pascal program and randomly change/delete/insert a character, there is a very high probability that you will have an invalid program. There is also a high probability that this invalidity can be detected by the compiler at compile time. Ie. Pascal will catch typos.

One of the few exceptions involves the semicolon. Randomly sprinkling semicolons can change the intent of a program as described in the User Manual and Report on page 26.

```
if p then; begin S1; S2; S3 end     is a surprise
```

To get around this problem (and to force everyone to use the semicolon as a separator instead of a terminator as God intended), I suggest making the empty statement invalid. In its place we would invent the null statement.

The null statement would make programs easier to read:

```
if p then null else S1

case n of
2: x:=5;
3: null;
4: x:=6
end
```

Other than that, Pascal is perfect and should be left alone. However, why not let people extend the language in any way they want -- by using pre-processors written in Pascal that produce Pascal code. Now all we need is an ingenious general purpose pre-processor to implement any goody your heart desires.

Love
Roedy

Roedy Green

The UNIVERSITY of WISCONSIN- LA CROSSE

LA CROSSE, WISCONSIN 54601 (608) 785-8000
785-8029

UNIVERSITY COMPUTER CENTER
JOHN C. STORLIE, DIRECTOR
HARVEY FOSSEN, ADMINISTRATIVE SERVICES
JOHN NIERENGARTEN, ACADEMIC SERVICES

July 2, 1979

Mr. Andy Mickel
Pascal User's Group
University Computer Center: 227 EX
208 SE Union Street
University of Minnesota
Minneapolis, Minnesota 55455

Dear Andy:

Per your request for information on what we're doing here with Pascal, I have the following.

We have a Hewlett Packard 3000 computer system which among other things supports undergraduate computer science instruction. In the past six months we have installed the contributed compiler from HP labs made available to the HP General Systems Users Group. The current version is fairly complete, although it is somewhat slow because it is a P-code system, which first translates into SPL (system programming language) and then compiles and executes the SPL.

Nonetheless, for pedagogical reasons our computer science department is going to teach Pascal. In fall 1979 we will introduce Pascal to three sections of Computer Science 121, Programming in Algorithmic Languages, replacing FORTRAN. This will introduce about 100 Computer Science students a semester to it and will provide them with a tool which they will use through much of the rest of their curriculum. Pascal meets a long unfulfilled need here for a block structured, high level language for teaching which enables one to teach proper programming structure.

Sincerely,

John A. Nierengarten
Assistant Director
Computer Center

JAN:lh

c.c. J. Storlie

THE ROYAL COLLEGE OF PHYSICIANS AND SURGEONS OF CANADA · LE COLLÈGE ROYAL DES MÉDECINS ET CHIRURGIENS DU CANAD

THE R. S. McLAUGHLIN
EXAMINATION AND RESEARCH CENTRE

LE CENTRE D'EXAMENS
ET DE RECHERCHES R. S. McLAUGHLIN

PROTECTRICE: SA MAJESTÉ LA REINE
PATRON: HER MAJESTY THE QUEEN

25th October, 1979

Andy Mickel
Pascal Users' Group
University Computer Centre
208 SE Union Street
University of Minnesota
Minneapolis. MN 55455

Dear Andy (-if I may?),

Thank you for returning my call yesterday regarding the small print size of the PUG Newsletter.

I find your reply that the reduced print size will continue disappointing, of course.

Your remark to my secretary that you have only had about four or five complaints about print size is of uncertain value as an argument. How many people, disgusted by the print size, did not trouble to call? Bearing in mind that your distribution is world-wide, many people rather distant from Minneapolis might be slightly more reluctant to call than I was; I very nearly did not call.

I am always rather disturbed at the insistence on uniformity in the name of technology, or efficiency, or cost. Surely people should come first? Why not leave a few print-outs at full size, and ship those to the feeble-sighted? You save on the cost of reducing and binding, at the expense of a little extra organization.

I do appreciate that yours is a volunteer effort, run with minimal staff. But are not PASCAL and its devotees worth it?

Yours sincerely,

Colin Park (Ph. D)
(Assistant Director).

P.S. In the case of PASCAL, some of us may even be prepared to pay a little more for the privilege of not straining our eyes.

8-113 CLINICAL SCIENCES BLDG. UNIVERSITY OF ALBERTA, EDMONTON, CANADA T6G 2G3

Pete Goodeve
3012 Deakin Street, apt D
Berkeley, Calif. 94705

1979 November 9

Andy Mickel
Pascal News
University Computer Center
227 Experimental Engineering Building
208 Southeast Union Street
University of Minnesota
Minneapolis, Minnesota 55455

Dear Andy:

Willett Kempton mentioned, in his letter to you of a few months back, that I was finishing up a new Pascal system for Data General AOS installations, and ever since then I have been getting around to sending you a proper report. As the system has been stable now for over a month, it is obviously high time to finally get this note written.

We have actually had a version of AOS Pascal out in the field for nearly a year now, but this is basically the Lancaster P4 Nova RDOS Pascal with a run-time system modified to mate with AOS. The new edition is extensively rewritten, at both the run time and compiler levels.

The run time interpeter now takes full advantage of the Eclipse's instruction set (rather than being Nova compatible), has completely revised file-variable management and has been expunged of the few existing (and actually rarely encountered) bugs in the Lancaster original.

The compiler is now considerably closer to the (draft) standard than is P4: I had initially hoped to remove all the essential discrepancies, but a couple still remain due to time and budget limits. A couple of non-standard features -- in the form of some additional predeclared procedures (modified from the Lancaster original) -- improve the links to the external world somewhat. These are: a) abnormal program termination with HALT; b) random access of the components of any Pascal file via GETRANDOM and PUTRANDOM. This compiler -- like its Lancaster parent -- supports external procedure declarations, and as these may be written in either Pascal or assembly language, the user has considerable freedom in adding any system functions etc. that may suit him.

I should point out that what Lancaster calls "P4" has been considerably extended from the original Zurich version. In particular, it embodies full, typed file-variable facilities, including external files. I have had the gall to label the new compiler "P5" to avoid some of such guilt by association. Restrictions that have now disappeared include:

1) Upper-case-only ASCII: lower case may now be used freely in program text; it is not distinguished from its upper case equivalent. The standard brace convention for comments is allowed.

2) Tiny string constants (originally 16 chars max): the limit has been (arbitrarily) extended to 120 chars, but compiler heap space used corresponds to actual length.

3) GOTOs within procedures only: the full Pascal standard is now implemented; this was felt to be important for the occasionally vital "panic sequence".

4) No second field-width specifier for real output: full standard formatting is now implemented.

Other changes to the compiler -- such as increased set-size -- are really only relevant to this implementation, and I will leave them aside here, but one other internal change may be of more general interest. It turns out that while the stack frame size allocation mechanism used in the original P4 was quite adequate for an implementation where all stack elements are the same size, it doesn't really cope with the situation of differing sizes. In brief, when generating a P-code instruction that does not have a fixed operand type, the compiler didn't take the actual type into account when allocating space on the stack; instead, it would allocate the largest possible size if the instruction was a "push" type, and release the smallest possible in the case of a "pop". This meant that the longer the procedure, the larger the stack frame it apparently would need, while in fact most procedures really need very little in the way of temporary space. This defect became especially severe when we went to 8-word sets! The P5 algorithm is exact, keeping proper track of the amount of space needed or released by each instruction.

Like a number of other systems around, the approach to generating an executable Pascal program is for the compiler to generate a fairly low-level symbolic "P-code" from the original source; this is converted to binary form and bound with the run-time library modules to create an executable file; the whole sequence of course follows automatically from a single command to the operating system by the user. I don't intend to get into discussion here of the relative merits of interpretation versus compilation to machine code, although the system seems to perform very creditably against DG Fortran, for instance. The main advantage of this approach as I see it is its modularity: if one later wants true compiled code out of the system, there is no need to touch the compiler at all; P-code appears to translate very smoothly into many machine instruction sets (including that of the Eclipse) and in some cases this may be possible using an

existing macroassembler. (In fact, for simplicity and because of the slowness of macro expansion, in our system even the translation of P-code to its packed form is mostly done by a translator written in Pascal.) Certainly, if the P-code is complete enough, it should be reasonably simple to produce translators and interpreters for different machines, using exactly the same compiler.

Because first Lancaster, and then ourselves, found some lacks in the Zurich P4 P-code in the ancillary information that one would like to have when generating a binary version of the code, an attempt has been made in the P5 variant to pass on all the information that a translator program might need, in a form entirely independent of the target machine. The P-code instructions and their formats are unchanged from the original, except for the inclusion of the new facilities, but a new statement type -- the "directive" -- has been added. Directives are used to indicate such things as procedure entry labels -- together with their original Pascal identifiers; this sort of extra information is useful in building "memory maps" or other debugging aids during translation. External procedure declarations and entry points also have their own directives, so that suitable links can be set up when the modules are bound into executable form. Other directives supply the program name and so on, and the source line numbers now appear with the instruction counts recorded in the P-code.

I had intended to enclose a specification sheet for the Implementation Notes of the "News", but I think we should be sure that you receive it in final released form, so I will let Gamma Technology supply that item. If anyone is interested in more details in the meantime, they are welcome to contact:

GAMMA TECHNOLOGY, INC.
2452 Embarcadero Way
Palo Alto, Calif. 94705
(415) 856-7421

Sincerely,

Rti

PATTERN ANALYSIS & RECOGNITION CORP.

228 LIBERTY PLAZA
ROME, N.Y. 13440
TEL. 315-336-8400

15 February 1979

Mr. Timothy M. Bonham
D605/1630 S. Sixth Street
Minneapolis, MN 55454

Dear Tim:

I have modified the PDP-11 pascal compiler kit (version 4) distributed by DECUS and by Seved Torsterdahl (see Pascal News #12, June 1978) to improve it in several ways and would like to make it available to interested RSX-IAS users. I have called my modified kit version 4.5, to avoid confusion, because version 5 is now available from DECUS. All of the modifications were made in order to allow the compiler to compile itself (until now it had to be cross-compiled using a DEC-10), but as a side effect my version has the following advantages:

1) Can be configured to have one of three different levels of overlaying (with correspondingly different symbol table space) in order to allow trading of compilation speed for capacity to compile large programs.

2) When configured with lightest overlaying, overlay swapping time is minimal and compiler runs three times faster than version 4.

3) Produces object code which is 12% smaller than and is faster than version 4.

4) I corrected bugs to allow procedural parameters to work.

5) It can compile itself in approximately 15 minutes (without using memory resident overlays) with all files on the same RPØ6 disk drive.

Persons interested in obtaining a copy should contact Richard Cichelli or John Iobst, who will be distributing the kit (and making further fixes and improvements) at the following address:

A.N.P.A. Research Institute
1350 Sullivan Trail
P.O. Box 598
Easton, PA 18042

Sincerely,

Michael N. Condict

Michael N. Condict

MC/dms

KF
EDP department
Michael Evans

Datum 1978-10-26

Beteckning

PROBLEMS IMPLEMENTING PASCAL IN A COMMERCIAL ENVIRONMENT

We are interested in implementing Pascal as a normal programming language in parallel with COBOL and Assembler. The current program development environment is -

- IBM 370/168 under MVS
- Interactive development using TSO
- Logical modular programming
- Interactive testing of modules
- Structured programming using macro COBOL (MetaCOBOL)
- Data base management using System 2000
- Applications development staff of about 70 persons

In order to be able to use Pascal in a production environment, we need to know about the future of Pascal in the following areas -

- Standardisation/Formalisation
- Integration with existing systems
- Special commercial requirements
- Development environment

Standardisation

Programs produced in our environment have long useful life times, up to 10 years. Before committing to a new language, we must be sure that it is going to survive that long.

Usually this kind of guarantee is provided by a machine supplier who undertakes support of a number of main line languages. Pascal is not one of our supplier's main line languages.

Another guarantee is given by a formal standardisation through ISO/ANSI. Pascal is in practice formalised via Wirth & Jensens book. More recently, the Pascal group at UCSD have taken on collection of Pascal extensions and modifications. Are all Pascal implementors going to accept and implement all extensions or is there going to be a foundation Pascal with many different extensions?

Various Pascal-like languages have been developed and are being developed. How much invested development must be scrapped if it turns out that one of these languages, for example ironman/DOD1, turns out to be a standard? How easy will it be to automatically convert to the new language. We have

KF
EDP department
Michael Evans

Datum 1978-10-26

Beteckning

converted between various COBOL dialects without leaving the COBOL language. It ought to be possible to convert to a new Pascal dialect if this does not involve a complete rethink regarding education, programming techniques, development tools etc.

One of the advantages of Pascal is the use of machine independent p-code. Is this standardized sufficiently that code from one compiler may be used with another machine which supports the same level of p-code? This is of interest for us as we envisage the use of satellite machines of various sizes with centrally developed programs. This development would be eased if tested object code could be sent to remote sites.

Pascal is taught at many universities. Unfortunately, many of our programmers, and many of those whom we employ in the future, have not had the benefit of this education. Are educational materials, in the form of video cassettes, course books, examples of good programming practices available for Pascal? It would be of great interest if we could get in touch with other installations, especially in Europe, who use Pascal in a commercial environment.

Integration

Although Pascal can be used to implement operating systems and data base management systems, these functions are normally already present in the commercial environment. Data already exists in some form of data base which must be accessed in a particular way, common functions such as date calculation are already implemented in standard program modules etc. For Pascal to be fully useable, it must be able to communicate with modules written in other languages. This communication must include being invoked by other modules (IMS calls to data base programs) and invoking other modules (IMS data base services). In addition, it is often suitable to divide an application into a number of separately compilable modules. Pascal must therefore be able to communicate with Pascal modules compiled on other occasions.

The same data structures are often used in a number of programs. To be improve safety and simplify development it would be useful to use the same physical definition. Some kind of source library management feature with a compiler directing COPY function is needed in Pascal.

Programming can be simplified if common functions are already coded and tested. The number of such common functions can become enormous if all combinations of parameter types are

KF
EDP department
Michael Evans

to be catered for. This problem may be avoided if a standard type "THING" were available. A parameter defined as THING may contain any type of data. It may only be used as a parameter to the standard function DATATYPE (variable) which returns BOOLEAN INTEGER REAL CHAR USER etc, or in an assignment statement. Execution time type checking would be needed in that statement but nowhere else. This admittedly breaks the rules of Pascal as a strongly typed language in the same way as GO TO breaks the rules of control structures. The type violation would however be well marked both in the invoked function and probably in the invoking function (in order to pass type information). It would allow such functions as general interfaces to external systems to be implemented in Pascal.

Pascal as defined by Jensen and Wirth only defines sequential files. It is often necessary to be able to access a particular record in a file, either by means of a key (indexed files) or by means of record number (relative or direct files). The use of Pascal would be eased if it were possible to program this in Pascal and not need Assembler routines to do it.

Other programming languages use different formats for internal data. These formats are often used on existing data files. It must be possible to access even these kinds of data. One method is to implement general Pascal functions to perform the conversion to and from standard Pascal types. To ease the coding of this function, the data type THING mentioned above would be useful. The other method would be to support data types which are already supported in FORTRAN/COBOL/PL1 even in Pascal possibly with some limitations.

Commercial programming

Arithmetic operations often involve a fixed number of decimal places. It must be possible to define these fields as integer with decimal shift instead of risking inaccuracy caused by floating point errors.

The formatting requirements for figures in a financial listing are many and varied. Zero suppression, credit/debit signalling, thousand comma insertion and floating currency sign are just a few of the features need. Pascal must be able to define the editing required when outputting numeric variables to text files in a way similar to COBOL's report item PICTURE clause. If this is not done centrally, each implementor will find his own way of editing, resulting in confusion similar to that surrounding BASIC's PRINT USING statement.

KF
EDP department
Michael Evans

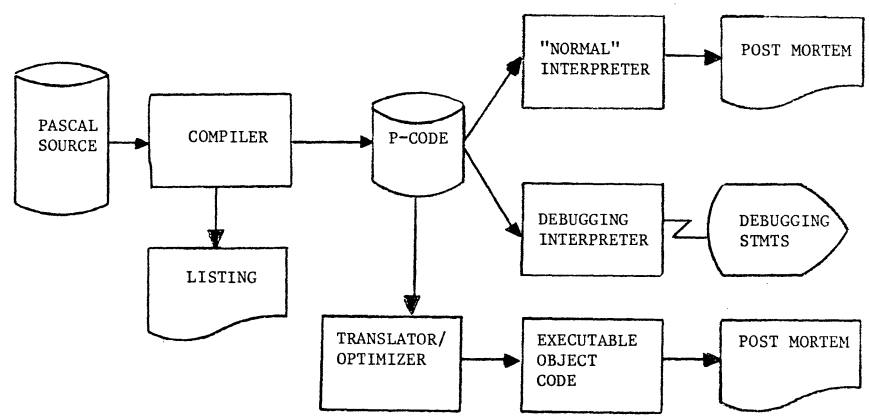
Datum 1978-10-26 Beteckning

Development environment

The programmers involved in developing commercial programs are among the most expensive items used during development. To enable them to work as efficiently as possible, they must have better tools than a straight batch compiler. Combined with a suitable compiler, Pascal p-code gives the ability to have an advanced interpreter which allows single statement execution, breakpoints, setting och listing global and local variables, statement trace and path execution summaries. When used in an interactive environment, these features would greatly ease program development. Many of these ideas are found in the current Pascal compiler series in Byte magazine.

As Pascal makes it easier to write large compilable units than many separately compilable modules, it is essential that the programmer be able to find variable definitions and uses. A cross reference listing would be very useful.

Occasionally, commercial data processing requires handling of such large data volumes that speed of operation is a critical issue. To allow Pascal to be used in these situations, it must be possible to translate p-code to executable machine instructions on the target machine. It may even be possible to optimize this machine code. The interrelationship between these functions is illustrated below.



Post-script

This paper contains the ramblings of a newcomer to the Pascal user community. I have tried to see Pascal through the eyes of the business data processing department where I am responsible for programming methodology. I imagine that many of the questions have been answered earlier or rejected as contrary to the spirit of Pascal, in which case I apologize.

az ☆ ☆ ☆ ☆ ☆ ☆ ☆ ☆

Pascal Standards Progress Report

Jim Miner, 1 December 1979

Several newsworthy events have occurred since the last Progress Report in Pascal News #15 (pages 90-95). In a nutshell, these events show substantial progress toward an international Pascal standard. (See the Progress Report in #15 for a glossary of some the terms used here.)

Another Working Draft

As expected in the last Progress Report, a fourth working draft prepared by BSI DPS/13/4 was distributed within standards organizations in October by the secretariat of ISO/TC 97/SC 5 under the document number "N510". (Recall that the previous draft is called "N462".) N510 contains a large number of changes from N462. Most of these changes are corrections to "obvious" errors and oversights. A smaller number of changes address fundamental ambiguities or other technical flaws in the Pascal User Manual and Report; these changes often are more controversial than the "obvious" ones.

As an example of the more controversial kind of change, consider the restrictions placed on labels to which goto-statements may refer. The User Manual and Report specifies that a goto may not jump into a structured statement. Although the wording in N462 was felt to be unclear, this was the intent of the restriction in that document. But the comments received from the public on this section of the draft clearly showed that run-time tests were required to enforce the restriction in the case of goto's which jump out of procedures or functions. In order to allow efficient compile-time checking of goto restrictions, the restrictions were tightened in N510, as described later. At the same time, of course, the wording was clarified in response to many comments.

A full list of all changes between N510 and N462 would be very difficult to compile and explain. Rather, we hope to print in a future issue of Pascal News the next draft which will be based on N510.

However, there is one very important new language feature which was introduced in N510. This feature is called "conformant array parameters". The feature was added to N510 in response to the many comments, including those from Niklaus Wirth and Tony Hoare, which cited as a major shortcoming in Pascal the inability to substitute arrays of different sizes for a given formal parameter in procedure and function calls. Because this such a significant and recent change, Arthur Sale has written the description which appears below.

The Experts Group Meeting

The new draft, N510, served as a basis for discussion at the meeting on November 12 and 13 in Turin Italy of the ad hoc Experts Group. This meeting was held in conjuction with the ISO/TC 97/SC 5 meeting on November 14..16. The following individuals were in attendance.

Franco Sirovich (Italy)	David Jones (USA)
Bill Price (USA)	Coen Bron (Netherlands)
Michel Gien (France)	Jim Miner (USA)
Christian Craff (France)	Scott Jameson (USA)
Olivier Lecarme (France)	Makoto Yoshioka (Japan)
William A. Whitaker (USA)	Akio Aoyama (Japan)
Don MacLaren (USA)	Albrecht Biedl (Germany)
Fidelis Umeh (USA)	Arthur Sale (Australia)
Bengt Cedheim (Sweden)	Emile Hazan (France)
Marius Troost (USA)	Tony Addyman (UK)

The purpose of the meeting was twofold: first, to advise the "sponsoring body" (BSI, represented by Tony Addyman) on solutions for remaining technical issues, and, second, to advise SC5 on a course of action for further work on the standard. Most of the two days was spent on technical issues.

Technical issues were informally divided into three categories: (1) "niggles" (or "obvious" problems having fairly simple solutions), (2) "local" issues which affected few sections of the draft, and (3) issues of greater magnitude, affecting several sections of the draft. Naturally, discussion centered on the last two categories.

An example of a "local" issue (category 2) was mentioned above, namely the restrictions on labels and goto's. The relevant section reads as follows.

> 6.8 Statements
>
> 6.8.1 General. Statements shall denote algorithmic actions, and shall be executable. They may be prefixed by a label. Within its scope, a label shall only be used in the statement S that it prefixes, the conditional-statement (if any) of which S is an immediate constituent, the statement-sequence (if any) of which S is an immediate constituent, and, if this statement-sequence is the statement-part of a block, the procedure-declarations and function-declarations of that block.
>
> statement = [label ":"] (simple-statement | structured-statement) .

The group quickly agreed both that the word "used" (in "a label shall only be used") should be changed to indicate the fact that the only possible use of a label is a reference by a goto-statement, and also that the long sentence which states the restrictions on references to labels should be broken down into more-easily understood parts. It was agreed that the intent of the sentence allows goto-statements to reference the label of a statement S only in the following contexts:

(a) when the goto-statement occurs anywhere within S; or

(b) when the goto-statement occurs anywhere within the if-statement or case-statement of which S is one "branch" or component statement (e.g., the goto may occur anywhere in the else part of an if-statement and still reference the label on the then statement, but not a label within it); or

(c) when the goto-statement occurs anywhere within a statement-sequence (in a compound-statement or a repeat-statement) of which S is a component statement; or

(d) when the goto-statement occurs in a procedure or function declaration (within the scope of the label) nested in the block which declares the label (i.e., non-local goto's), and only if the statement prefixed by the label is not nested inside a structured statement (other than the compound-statement which is the statement part of the block).

More than one member of the group certified that these restrictions can be enforced efficiently by a one-pass compiler with no run-time overhead.

There was resistance to allowing jumps between "branches" of conditional-statements. It was argued that the use of such goto's is not good, is poor "style", and should not be part of the standard. After some discussion, the group agreed to further restrict the goto by not allowing the references cited in (b), above. As with most such changes, the precise redrafting was left to an individual member of the group.

The major topic of discussion was conformant array parameters. This was confused by the fact that the form present in N510 had already been renounced by the British in favor of a form drafted by Arthur Sale. With the exception of a different proposal by Coen Bron, which was closer to the version in N510, there was nearly unanimous support for the version proposed by Arthur Sale. (See his note, below, for a description.)

In addition to technical issues, the Experts Group also briefly reviewed Pascal standards activities within the nations represented. It was clear that the approach taken by BSI DPS/13/4 toward a Pascal standard had a great deal of support internationally, with the exception of a few technical details. Discussions are currently underway in attempting to resolve those issues not completed in Turin.

Also evident was significant interest in extensions to Pascal within several countries, especially France, Germany, the Netherlands, and the United States. Therefore, any future extended standard must be developed in cooperation between the interested national groups as a longer-range project. This project undoubtedly will involve the newly-formed Working Group discussed below.

The SC5 Plenary Session

The Experts Group sent two resolutions to SC5 for approval. The first resolution, which passed SC5 without opposition, states that Tony Addyman should revise the Pascal draft (N510) according to the agreements reached by the Experts Group, and that this revised draft would be registered as a Draft Proposal for voting. What this means is that some time in the next few months the revised draft will be distributed to SC5 voting members for a three-month letter ballot. We hope to print the full text of the Draft Proposal in Pascal News when it becomes available so that readers will have a chance to provide comments on it to their own national standards group.

The second resolution, passed unanimously by SC5, established a formal Working Group ("Working Group 4, Pascal") to advise the British group on further standardization, and to consider proposals for such from bodies recognized by ISO. The Working Group is intended to replace the Experts Group, and will be under the Convenorship of Tony Addyman. Members are to be nominated by SC5 member bodies. This group will aid in resolving negative comments (if any) on the new Draft Proposal, and will probably coordinate future work on Pascal extensions.

The SC5 meeting also saw an interesting exchange on the subject of Ada (the U.S. Department of Defense language). William A. Whitaker, attending as an observer from the United States, made a presentation to SC5 on Ada. Under questioning by the Australian representative (Arthur Sale), Whitaker admitted that Ada actually has little in common with Pascal. This stands in stark contrast to the impression one might get from reading DoD press releases and other articles which some feel have attempted to lend credence to the Ada project by associating it with Pascal. Thankfully, Pascal need no longer suffer from such derogatory associations!

In the United States

Several small points should be noted as having changed since the Progress Report in #15. These changes occurred at the meeting of November 28..30 in Boston.

First, a single joint committee has been formed from the ANSI-X3J9 and the IEEE Pascal Standards Committees. The new committee is formally called the "Joint ANSI/X3J9 - IEEE Pascal Standards Committee", abbreviated "JPC".

Second, Jess Irwin has resigned as secretary of X3J9. Carol Sledge of On-Line Systems has volunteered to take the job. Correspondence with the JPC should be sent to:

> Carol Sledge (X3J9)
> c/o X3 Secretariat
> CBEMA: Suite 1200
> 1828 L Street NW
> Washington, D.C. 20036

Third, the proposed "SD-3" for considering extensions to Pascal printed in #15 (pages 93..95) was modified to reflect the international interest in Pascal extensions which was apparent at the Turin meeting. The revised document specifies that the JPC will cooperate with Working Group 4 of SC5 on developing an international extended standard, and that the resulting American National Standard will be compatible.

****

Implementation Notes

Editorial

Tektronix
COMMITTED TO EXCELLENCE

Tektronix, inc.
P.O. Box 500
Beaverton, Oregon 97077

Phone: (503) 644-0161
TWX: 910-467-8708

First, the formalities:

Bob Dietrich
MS 63-211
Tektronix, Inc. phone: (503) 682-3411 ext 3018
P.O. Box 500
Beaverton, Oregon 97077
U.S.A.

Feel free to call me (I'm usually in between 10AM and 5PM Pacific time), but consider yourself lucky if you find me near the phone. I don't have a secretary, and may have to be paged. Consider yourself foolhardy if you write me and expect a personal reply in less than a year. I'll try to do better for those outside the U.S. Should you wish information on a specific implementation, please read Pascal News first. It's unlikely that I will have any more current information than can be found there. Furthermore, this can put me in the delicate position of seeming to endorse a particular implementation, which I will not do for ethical and legal reasons. These cautions aside, I'll do what I can to help.

Next,the traditional goals statement of a new editor. At this time, I don't plan to change anything (I know - every new editor and politician says the same thing). My basic goal is to publish a comprehensive list of Pascal implementations by the summer of 1980. Whether this will appear in one issue or several is yet to be discovered: a great deal depends on the cooperation of the readers of Pascal News. Which brings me to the next topic.

As you may have noticed, the Implementation Notes section was pretty sparse in issue # 15, and almost non-existent in this issue (except for a long winded editorial). The reason is we have received very few new reports and/or updates of implementations in recent months. No garbage in, no garbage out. To remedy this problem, I will be mounting a mail campaign as has been done in previous years. Anyone and everyone who has ever even hinted they had an implementation of Pascal will be getting a letter requesting a new implementation checklist. However, I do realize how difficult it is at times to answer mail. To save us both some trouble, you will notice a brand new Pascal News One Purpose Coupon at the back of this issue (not to be confused with the ALL Purpose Coupon). This amazing piece of paper is simply an implementation checklist with room to write on. Fill in the blanks, fold, and mail to the address on the back. Feel free to also send in camera ready checklists. I hope this will give us a little more to print in the meantime.

Now for my biggest irritation. In my everyday work, I have used many different implementations/versions of Pascal. On our DECsystem-10 alone we have six different versions of Pascal available. This does not include some cross-compilers for other machines. Why do we have so many different compilers for the same machine?

The reason we have so many versions still active is that many user programs have not been updated to account for major changes in new releases of the compilers, and so the old release stays around. Most of the changes have been non-trivial, and heavily impact whether the programs can be simply recompiled under the new release. The changes have included the way the character set, terminal I/O, I/O in general, operating system interface, et cetera, are mishandled. Even worse are the versions that made "improvements" to the language, such as:

```
j := case k+34 of ...
```

And

```
l := if FouledUp then sqr(x) else sqrt(z)
```

Of course, the changes are rarely upward or cross compatible.

The root of the problem, and the part that irritates me, is the fact that the compilers implement different extended subsets of Pascal. This means they implement entirely different languages, not Pascal. None of these compilers implement all of "standard" Pascal (as in the Jensen and Wirth Report); however, all the versions have been "extended" in quite a few arbitrary ways. Little attention seems to have been given to eradicating errors, even those due to the P2 heritage of the compilers. In fact, one of the versions came out with quite a few extensions (many of them bastardizations) and none of the errors of its predecessor corrected. (In all fairness, two of the versions have had major error corrections performed on them). These shortcomings and extensions make it very difficult for programs to be transported both to and from our installation, especially for those not totally aware of the problem.

Please do not misunderstand me. I am not against extending Pascal (well, at least not totally against it). Some extensions make the bootstrap process for a new implementation much easier. What I am against is the effort put into extensions that would be much better

directed toward fixing errors and implementing the full language. How I long for a full set of {ASCII} char! As a compiler writer, I realize that extending the language in a particular implementation both is fun and might help differentiate the product in the marketplace. This is especially true in the Pascal market, where both users and implementors have not really understood the language.

What I am really looking for (I don't think I'm alone) is quality in the tools I use. Just as I wouldn't be too happy buying a saw missing half its teeth, or with half the teeth backwards, or with teeth on the handle where my hand is supposed to go, I really dislike the so-called implementations of Pascal whose manuals list ten major omissions to the language and thirty "improvements". The omissions are even less tolerable once an implementation has gotten past the "well, at least it looks like Pascal" stage. This is not quality.

Perhaps you feel I expect too much for a language that owes its popularity to the efforts of many individuals rather than large companies. True, we owe these implementors a debt that will never be repaid. But this debt does not relieve implementors of their responsibility to do the job right, especially if they have the time and energy to make their own "improvements" to the language. Another reason I expect quality is for the many new users of Pascal. These users judge the language itself by the particular implementation they are first exposed to, and I have already seen some discouraged by poor implementations. The most important reason to hold implementors responsible for quality is the simple fact that if we do not, there won't be any, and we the users will find it much more difficult to get our own jobs done.

A good many implementors are professional enough to assume this responsibility for quality, and have probably already done so. What of those individuals and companies who have not? What can we users do? Well, the best approach is to convince the implementor that conforming to standard Pascal is in the implementor's own best interests. The reasons can be many: good will, conditions of purchase, additional sales, blackmail, advertising, even legal requirements. In many countries, adoption of a standard (such as ISO Pascal) gives it the weight of law. Any product purporting to be Pascal in such a country MUST conform to the standard.

It is fortunate that there are now two tools to back up this demand for quality. The first, of course, is the upcoming ISO Pascal standard. There are admittedly problems because the standard is not yet official, but at the same time the standard is for the most part not all that different from the Jensen and Wirth Report. Getting most implementations to conform to the Report would be a major accomplishment in itself, and not that far from where the ISO standard will probably end up.

The second tool to help quality is the Pascal Validation Suite that was published in issue # 16. The biggest problem in quality assurance is finding quality tests, and the validation suite goes a long way toward solving this problem. It is also a very big advantage to have the suite available now, even before the Pascal standard is adopted. Implementors of other language standards had to wait quite a while (many are still waiting) before such a measurement device was available. I will have quite a bit more confidence in a particular implementation of Pascal if I know the results of having it try to process the validation suite.

I would like to encourage both users and implementors to use the validation suite and send the results to me as well as to Arthur Sale. By all means, also send a copy to the implementor. I will then publish the reports I receive in Pascal News for the world to see. Please see the sample reports in issue # 16 for format. I would hope that over the next year we can get reports for each and every implementation (then again, I always have been an optimist). The letter campaign to implementors will also be requesting reports of the validation suite results.

One last comment. Be kind to your implementor, especially if he is doing a good job. It's not all that easy to wrestle many of our poorly designed machines into speaking Pascal. Don't use the validation suite to beat him senseless, but have some patience. On the other hand, if he has implemented something that cannot even pass for a subset of Pascal, cannot add two numbers correctly, and has a lot of "improvements", be merciless.

Implementation Critiques

Digital Equipment PDP-11 ('Swedish')

1979 December 19

A critique of the Swedish Pascal compiler

(as derived from its User Manual by A.H.J.Sale)

1. The User Manual is a supplement to Jensen & Wirth. It is well-written, and describes the implementation of Seved Torstendahl running on PDP-11s under RSX-11M and IAS.

2. The manual first describes how to run Pascal programs under the operating systems, how to attach files, etc.

3. The next section addresses extensions. The tokens are extended by:

(.	for	[	} reasonable, given poor print capability
.)		]	and use of these ISO positions for Swedish chars.
(*		{	} not extension; allowed by standard.
*)		}	
#	for	<>	} These three extensions are simply undesirable;
&	for	and	especially since neither # nor ! capture
!	for	or	any of the meaning of the well-defined tokens.

I recommend the removal of the last three extensions as being contrary to the best interests of portability of programs and programming skills, and ugly as well.

4. The document introduces extra pre-defined constants: MAXINT, MININT, ALFALENG, MAXREAL, MINREAL, and SMALLREAL. It mis-calls these 'standard' constants which they are not, except for MAXINT. No problems with the introduction of extra constants provided they are properly identified.

5. 'Standard' types is also misused. TEXT is indeed standard and need not be in a section on extensions, but types

```
ASCII = CHR(0)..CHR(127);  BYTE = CHR(0)..CHR(255);
```

are simply extra pre-defined types. This misuse of 'standard' runs throughout the document. Something can only be called 'standard' if it conforms to a standard, either the *old de facto* standard of Jensen & Wirth, or preferably the new ISO draft standard. We may as well get this right now.

COMPUTER STUDIES GROUP

The University of Southampton

6. The extended case statement (otherwise clause) does not use the syntax more or less agreed internationally and published in Pascal News, but uses an OTHERS label. The syntax suggests it need not be last.

7. A LOOP-END construct is introduced, together with an EXIT. I strongly recommend the removal of this construct which is a frequent cause of error in many programs. Why it was introduced is difficult to understand since Pascal handles the so-called loop-and-a-half structure much better without it.

8. There are 'Standard' procedures DATE and TIME; it seems a pity that these cannot be guaranteed to relate to the same instant, and that a single TIMESTAMP cannot return both values guaranteed synchronous. NEW is implemented, but not DISPOSE; MARK and RELEASE are provided.

   There is a HALT, and RESET and REWRITE allow file selection by additional parameters. BREAK flushes line buffers for the special file TTY, and acts as WRITELN for other text-files (irregular). PAGE inserts a form-feed character into the text-file.

   Random access is provided by allowing another integer parameter to GET and PUT. I cannot understand why people prefer to overload names with new meaning and introduce irregularity in preference to choosing new names such as PUTR and GETR. Especially since the axioms of GET/PUT do not hold.

9. There are additional 'standard' functions RUNTIME, TWOPOW, SPLITREAL and IORESULT.

10. There is an adjustable array parameter feature. How it works is a mystery as the component-type is apparently not given. The following example is taken from the manual:

```
PROCEDURE MATADD(VAR A,B,C: ARRAY[INTEGER,INTEGER]);
```

It would seem highly desirable to alter this implementation to something with more abstract structure, and more checkable.

11. There is also a facility to declare a new kind of parameter

```
PROCEDURE PRINT(STRING S);
```

This feature turns Pascal's ordering on its head (type precedes identifier) and it misuses the word 'string' by defining it to be an array! The facility is badly expressed, and should be described in terms of a sequence (= file) of characters.

12. There is a facility to pass procedural and functional parameters, but it differs from the draft ISO standard in defining a new form of parameter-list. Congruity of two parameter lists is not adequately defined, but this is an informal document.

13. There is an external compilation facility; the directive EXTERN is used followed by a parameter list. Examples

    EXTERN(FORTRAN)

    or EXTERN(FORTRAN,'TEST')

14. The reserved word list is extended by LOOP, EXIT, OTHERS, EXTERN. If the loop construct is removed this drops to two.

15. PACK and UNPACK are not implemented; only char and Boolean arrays are packed. (Presumably no records are packed, which is very unsatisfactory for many mini and micro applications.)

16. Only local GOTOs are permitted; a set may have up to 64 elements; files may only be declared in the main program.

17. The documentation cheats on MAXINT by disallowing it as a limit in a for-loop. It would be accurate to say that MAXINT in this implementation is really 32766, and that the constant called MAXINT should be renamed, perhaps to BIGINT or similar. Or the implementation should be improved (see Pascal News 15).

18. Set of char is allowed, by defining the type char to be the subrange of characters from CHR(32)..CHR(95). Of course this violates a lot of Pascal axioms, notably about the type of the result of CHR. A very crude approach to the problem. It should be done right.

Arthur Sale

A.H.J.Sale

{ See Zilog Z-80 (Digital Marketing) }

{ See Zilog Z-80 (MetaTech) }

{ See GOLEM B (Weizmann) }

This compiler runs under CP/M and produces macro-assembler code. The price is $275.

Ithaca Intersystems (formerly Ithaca Audio)
1650 Hanshaw Road
P.O. Box 91
Ithaca, NY 14850

This compiler runs under CP/M and is a Pascal-P descendant. The price is $350.

Digital Marketing
2670 Cherry Lane
Walnut Creek, CA 994596

This is a compiler for a cassette-based system, and sells for $35.

Dynasoft Systems
POB 51
Windsor Junction, North Saskatchewan B0N 2V0
Canada

The information on this compiler is unclear. It appears to be all or partly in ROM, and sells for £40.

The Golden River Co., Ltd.
Telford Road.
Bicester, Oxfordshire OX6 0UL
England

Validation Suite Reports

The University of Tasmania

Postal Address: Box 252C, G.P.O., Hobart, Tasmania, Australia 7001

Telephone: 23 0561. Cables 'Tasuni' Telex: 58150 UNTAS

IN REPLY PLEASE QUOTE:

FILE NO. .....

IF TELEPHONING OR CALLING

ASK FOR .....

4th December, I979

Mr. R. Shaw,
Digital Equipment Corp.,
5775 Peachtree-Dunwoody Road,
Atlanta, Georgia 30342
U.S.A.

Dear Rick,

Enclosed is a copy of a report of the Pascal Validation Suite on a VAX-II Pascal system. The report was produced for us by Les Cooper at La Trobe University.

The Pascal system is a Field Test version and is not available generally at this stage. All errors have been reported back to DEC who presumably will fix them before the system is finally released. The report should be seen in this context. Nevertheless, it provides an insight into what the VAX compiler will be like when it is officially released early in I980.

Les Cooper says he will provide an up-to-date copy of the report after the compiler has been officially released.

Yours sincerely,

Roy Freak

Roy A. Freak,
Information Science Department

Digital Equipment VAX 11/780 (DEC)

VAX-11 Pascal - Tested At LaTrobe University
Pascal Validation Suite Report

Pascal Processor Identification

Computer: Digital Equipment Corporation VAX-11/780
Processor: VAX-11 Pascal Field Test version T0.1-68

Test Conditions

Tester: Les Cooper
Computer Centre
La Trobe University
Australia

Date: November 1979

Version: Validation Suite 2.2

Conformance Tests

Number of Tests Passed: 128
Number of Tests Failed: 9

Details of Failed Tests

Test 6.4.3.3-1 failed because an empty record containing a semi-colon produces a syntax error.
Test 6.3.3.3-4 failed because an attempt to redefine a tag field elsewhere in the declaration part produces syntax errors.
Test 6.4.3.5-1 failed because an attempt define a file of pointertype failed to compile.
Test 6.5.1-1 failed because an attempt to define a file of filetype failed to compile.
Tests 6.6.3.1-5, 6.6.3.4-2 failed to compile where they tried to pass a procedure with a formal parameter list as as formal parameter to another procedure.
Test 6.9.4-15 shows that a write which does not specify the file does not write on the default file after reset(output).

Deviance Tests

Number of Deviations Correctly Detected: 61
Number of tests showing true extensions: 4
Number of tests not detecting erroneous deviations: 18
Number of tests failed: 5

Details of Extensions

Test 6.1.5-6 shows that lower case e may be used in numbers.
Tests 6.8.3.9-9, 6.8.3.9-13, 6.8.3.9-14 show that the following may be used as the controlled variable in a for statement: intermediate non-local variable, formal parameter, global variable.

Details of Deviations not Detected

Test 6.1.2-1 shows that nil may be redefined.
Tests 6.2.2-4, 6.3-6, 6.4.1-3 show that a common scope error was not detected by the compiler.
Tests 6.4.5-2, 6.4.5-3, 6.4.5-4, 6.4.5-5 indicate that type compatibility is used with var parameter elements rather than enforcing identical types.
Test 6.6.2-5 shows the compiler permits a function declaration with no assignment to the function identifier.
Tests 6.8.2.4-2, 6.8.2.4-3, 6.8.2.4-4 show that a goto between branches of a statement is permitted.
Tests 6.8.3.9-2, 6.8.3.9-3, 6.8.3.9-4, 6.8.3.9-16 show that assignment to a for statement control variable is not detected.
Test 6.9.4-9 shows that zero and negative filed widths are allowed in write.

Details of Failed Tests

Test 6.6.3.6-2, 6.6.3.6-3, 6.6.3.6-4, 6.6.3.6-5 check the compatibility of parameter lists. They fail to compile where they use a procedure with a formal parameter list as a parameter to another procedure. Test 6.8.3.9-19 shows that insecurities have been introduced into for statements by allowing non-local control variables.

Error Handling

Number of Errors correctly detected: 14
Number of errors not detected: 33

Details of errors not detected

Tests 6.8.3.9-5, 6.8.3.9-6, 6.2.1-7 indicates that undefined values are not detected.
Tests 6.4.3.3-5, 6.4.3.3-6, 6.4.3.3-7, 6.4.3.3-8 indicate that no checking is performed on the tag filed of variant records.
An assignment to an empty record is not detected in test 6.4.3.3-12.
Tests 6.4.6-4, 6.4.6-5, 6.4.6-6, 6.4.6-7, 6.4.6-8, 6.5.3.2-1, 6.8.3.5-5, 6.8.3.5-6, 6.6.6.4-4, 6.6.6.4-5, 6.6.6.4-7 indicate that no bounds checking is performed on array subscripts, subranges, set operations, or case selectors. Note: The system default is run time checks off. If the tests had been compiled with checks on then the checking would have been done.
Tests 6.5.4-1 and 6.5.4-2 show that a poor error message is given when a nil pointer is dereferenced and when an undefined pointer is dereferenced.
Test 6.6.2-6 shows that, if there is no result assigned to a function, there is no run time error message.
Test 6.6.5.6-6, 6.6.5.6-7 show that it is possible to change the current file position while the buffer variable is an actual parameter to a procedure and whilst the buffer variable is an element of the record variable list of a with statement.
Test 6.6.5.3-3, 6.6.5.3-4, 6.6.5.3-5, 6.6.5.3-6 show that there is no errormessage when the following occur as the pointer parameter of dispose: nil, undefined pointer, variable which is currently an actual parameter, variable which is an element of the record variable list of a with statement.
Test 6.6.5.3-7, 6.6.5.3-8, 6.6.5.3-9 fail because no checks are inserted to check pointers after they have

been assigned a value using the variant form of new.
Test 6.8.3.9-17 show that two nested for statements may ahve the same controlled variable.

Implementation Defined

Number of tests correctly run: 9
Number of tests incorrectly handled: 0

Details of implementation dependence

Test 6.4.2.2-7 shows maxint to be 21474883647
Test 6.4.3.4-2 shows that a set of char is permitted.
Test 6.4.3.4-4 shows that there are 255 elements in a set.
Tests 6.7.2.3-2 and 6.7.2.3-3 show that Boolean expressions are fully evaluated.
Tests 6.8.2.2-1 and 6.1.2.2-2 show that the variable is selected before the expression is evaluated in an assignment statement.
Test 6.10-2 shows that a rewrite is allowed on file output.
Test 6.11-1 shows that alternate comment delimiters are implemented.
Tests 6.11-2, 6.11-3 show that equivalent symbol cannot be used for the standard reference representation for the up arrow, :, ;, :=, [,], and the arithmetic operators.
Test 6.9.4-5 shows that two digits are written in an exponent.
Test 6.9.4-11 shows the default field width to be integer 10, Boolean 16, real 16.

Quality Measurement

Number of tests run: 23
Number of tests incorrectly handled: 0

Results of tests

Test 5.2.2-1 shows that identifiers are not distinguished over their whole length.
Test 6.1.3-3 shows that there are 15 significant characters in an identifier.
Test 6.1.8-4 shows that no warning is given if a { or ; is detected in a comment.
Tests 6.2.1-8, 6.2.1-9, and 6.5.1-2 indicate that large lists of declarations may be used in each block.
Test 6.4.3.2-4 indicates that integer indextype is not permitted.
Test 6.4.3.3-9 show that variant fields of a record occupy the same space, using the declared order.
Test 6.4.3.4-5 (Warshall's algorithm) took 1.010 CPU seconds and 249 bytes on the VAX-11/780. Note: This was using the VAX default of no run time checking.
Test 6.8.3.5-2 shows that no warning is given for impossible cases in a case statement.
Test 6.8.3.5-8 shows that a large populated case is accepted.
Test 6.8.3.9-18 shows that the undefined value of a for statement controlled variable is left in the range of its type.
Tests 6.8.3.9-20, 6.8.3.10-7 show that at least 15 levels of nesting are allowed when dealing with for statments, with statements, and procedures.
Test 6.9.4-10 shows that the output buffer is flushed at end of job.

Extensions

Number of tests run: 1

Test 6.8.3.5-14 shows that otherwise is implemented though not with the same syntax as that adopted at the UCSD Pascal workshop in July 1978.

VAX-11/780 Pascal - Commentary on Results

The Validation suite has shown up quite a number of flaws in the compiler, as documented in the preceeding report. Of particular concern is the apparent philosophy that the run time checking should be off by default.

These tests were run using Field Test version T0.1-68 of the compiler. With luck (a lot), the problems found will all be fixed before the compiler is released.

DEC has been informed of the results of all the tests. They have been given run listings, etc. where necessary. The replies they send to me (when they arrive) will be included in this section of the report.

PASCAL VALIDATION SUITE REPORT

Pascal Processor Identification

Computer: Apple II

Processor: UCSD Pascal version II.1

Test Conditions

Tester: R.A. Freak

Date: January 1980

Validation Suite Version: 2.2

Conformance Tests

Number of tests passed: 116

Number of tests failed: 22 (13 basic causes)

Details of failed tests

Test 6.1.2-3 shows that identifiers and reserved words are not distinguished correctly over their whole length.

Test 6.2.2-1 produces an error in scope.

Tests 6.4.3.3-1, 6.4.3.3-3 and 6.8.2.1-1 fail because empty field lists or empty records are not allowed.

Test 6.4.3.3-4 indicates that a tag field definition is not local to the record definition.

Tests 6.4.3.5-1 and 6.5.1-1 fail because a file of pointers is not permitted, nor can a file be part of a record structure.

Tests 6.6.3.1-5, 6.6.3.4-1, 6.6.3.4-2 and 6.6.3.5-1 fail because the passing of procedures/functions as parameters has not been implemented.

Tests 6.6.5.2-3 and 6.6.5.2-5 fail because eof is not set on an empty file, nor is it set after a rewrite.

Test 6.6.5.3-2 fails because dispose has not been implemented.

Test 6.6.5.4-1 fails because the procedures pack and unpack have not been implemented.

Test 6.8.2.4-1 fails because non-local gotos are not permitted.

Test 6.8.3.5-4 fails because a sparse case statement will not compile. (There is a limit on the size of each procedure).

Test 6.8.3.9-1 fails because the assignment to a for statement control variable follows the evaluation of the first expression. Use of extreme values in a for statement produces an infinite loop (test 6.8.3.9-1).

Tests 6.9.4-4 and 6.9.4-7 fail because the writing of real values does not conform to the standard and the writing of boolean values is not permitted.

Deviance Tests

Number of deviations correctly detected: 56

Number of tests showing true extensions: 7 (4 actual extensions)

Number of tests not detecting erroneous deviations: 25 (12 basic causes)

Number of tests failed: 6 (2 basic causes)

Details of extensions:

Tests 6.1.7-6 and 6.4.5-11 show that strings are allowed to have bounds other than 1..n and that compatible strings can have different numbers of components.

Tests 6.8.3.9-9 and 6.8.3.9-14 indicate that the for-control variable does not have to be local to the immediately enclosing block.

Tests 6.10-1 and 6.10-3 show that file parameters are ignored and the predefined identifier output may be re-defined.

Test 6.10-4 shows that a program does not have to have a program statement.

Details of deviations not detected:

Test 6.1.2-1 shows that nil may be redefined.

Tests 6.1.7-11 and 6.4.3.2-5 show that a null string is accepted by the compiler and that strings may have other than a subrange of integers as bounds.

Test 6.2.1-5 shows that an unsited label is not detected.

Tests 6.2.2-4, 6.3-6 and 6.4.1-3 contain a common scope error which is not detected.

Tests 6.3-5 and 6.7.2.2-9 show that the unary operator, +, may be applied to non-arithmetic operands.

Tests 6.4.5-2, 6.4.5-3, 6.4.5-4, 6.4.5-5 and 64.5-13 show that identical compatibility is not enforced.

Test 6.6.2-5 shows that a function without an assignment to the function variable is not detected.

Tests 6.6.6.3-4 and 6.6.6.4-6 show that real parameters are allowed for the function succ and pred, while trunc and round can have integer parameters.

Tests 6.8.2.4-2 and 6.8.2.4-3 show that a goto between branches of a statement is permitted.

Tests 6.8.3.9-2 and 6.8.3.9-3, 6.8.3.9-4 and 6.8.3.9-16 show that a for-control variable may be altered in the range of the for statement.

Test 6.8.3.9-19 shows that nested for statements using the same control variable are not detected.

Test 6.9.4-9 shows that integers may be written using a negative format.

Details of failed tests:

Tests 6.6.3.5-2, 6.6.3.6-2, 6.6.3.6-3, 6.6.3.6-4 and 6.6.3.6-5 fail because the passing of procedures/functions as parameters has not been implemented.

Test 6.8.2.4-4 fails because non-local gotos have not been implemented.

Error handling:

Number of errors correctly detected: 14

Number of errors not detected: 28 (14 basic causes)

Number of tests failed: 4 (1 basic cause)

Details of errors not detected:

Test 6.2.1-7 shows that variables are initialized to what was previously left in memory.

Tests 6.4.3.3-5, 6.4.3.3-6, 6.4.3.3-7 and 6.4.3.3-8 indicate that no checking is performed on the tag field of variant records.

An assignment to an empty record is not detected in test 6.4.3.3-12.

Tests 6.4.6-7, 6.4.6-8 and 6.7.2.4-1 indicate that no bounds checking is performed on set operations and overlapping sets are not detected.

Tests 6.5.4-1 and 6.5.4-2 show that a nil pointer or an uninitialized pointer are not detected before use.

Test 6.6.2-6 shows that a function without an assignment to the function variable is not detected.

Test 6.6.5.2-1 shows that a put on an input file is not detected.

Test 6.6.5.2-2 shows that a get past eof is not detected.

Test 6.6.5.2-7 indicates that a file buffer variable can be altered illegally.

Tests 6.6.5.3-7, 6.6.5.3-8 and 6.6.5.3-9 fail because no checks are inserted to check pointers after they have been assigned a value using the variant form of new.

Tests 6.6.6.4-4, 6.6.6.4-5 and 6.6.6.4-7 indicate that no bounds checking is performed on the functions succ, pred or chr.

Tests 6.7.2.2-6 and 6.7.2.2-7 show that integer overflow and underflow conditions are not detected.

Tests 6.8.3.5-5 and 6.8.3.5-6 show that if the value of the case index does not correspond to a case label, control passes to the statement after the case statement. The error is not detected.

Tests 6.8.3.9-5, 6.8.3.9-6 and 6.8.3.9-17 show that a for control variable may be used after the for loop has terminated. Nested for loops using the same control variable are not detected.

Details of failed tests:

Tests 6.6.5.3-3, 6.6.5.3-4, 6.6.5.3-5 and 6.6.5.3-6 fail because dispose has not been implemented.

Implementationdefined

Number of tests run: 15

Number of tests incorrectly handled: 0

Details of implementation-definition:

Test 6.4.2.2-7 shows maxint to be 32767.

Tests 6.4.3.4-2 and 6.4.3.4-4 show that large sets are accepted by the compiler but a run-time limit of 512 elements is imposed. A set of char is allowed.

Test 6.6.6.1-1 shows that no standard procedures or functions may be passed as parameters.

Test 6.6.6.2-11 gives some details of real number formats and machine characteristics.

Tests 6.7.2.3-2 and 6.7.2.3-3 show that boolean expressions are fully evaluated.

Tests 6.8.2.2-1 and 6.8.2.2-2 show that a variable is selected before the expression is evaluated in an assignment statement.

Tests 6.9.4-5 and 6.9.4-11 show that the number of digits in an exponent field varies according to the size of the exponent. The default output field width for integers and reals also varies according to the size of the expression printed.

Test 6.10-2 indicates that a rewrite cannot be performed on the standard file, output.

Tests 6.11-1, 6.11-2 and 6.11-3 show that the alternative comment delimiters have been implemented but no other equivalent symbols have been implemented.

Quality Measurement

Number of tests run: 22

Number of tests incorrectly handled: 1

Results of tests:

Tests 5.2.2-1 and 6.1.3-3 show that identifiers are distinguished over their first eight characters only.

Test 6.1.8-4 indicates that no help is provided for detecting unclosed comments.

Tests 6.2.1-8 and 6.2.1-9 indicate that more than 50 types may be compiled and more than 50 labels may be declared and sited. Test 6.5.1-2 shows a limit of 70 identifiers in a list has been imposed.

Test 6.4.3.2-4 shows that an array with an integer index-type is not permitted.

Test 6.4.3.3-9 shows that variant fields of a recrod use reverse correlation for storage.

Test 6.4.3.4-5 (Warshall's algorithm) took 166 bytes of code. No timing information is available.

Test 6.6.1-7 shows that procedures may be nested to a depth of 7. For statements may be nested to a depth greater than 15 (test 6.8.3.9-20) but with statements may be nested to a depth of 11 (test 6.8.3.10-7).

Tests 6.6.6.2-6, 6.6.6.2-7, 6.6.62-8, 6 6.6.2-9 and 6.6.6.2-10 tested the sqrt, atan, exp, sin/cos and ln functions and all tests were completed successfully. (The tests had to be modified to avoid the limit placed on procedure size).

Test 6.7.2.2-4 shows that division into negative operands is inconsistent but division by negative operands is consisten. The quotient is trunc (A/B) for all operands. mod(a,b) lies in (0,b-1).

Test 6.8.3.5-2 shows that no warning is given for a case constant which cannot be reached.

Test 6.8.3.9-18 shows that no range checks are inserted on a for control variable after a for loop.

Test 6.9.4-10 shows that the file, output, is flushed at end of job and test 6.9.4-14 shows that recursive I/O using the same file is allowed.

Details of failed tests:

Test 6.8.3.5-8 fails - a large case statement causes the size of the procedure to overflow the maximum limit.

Extensions

Number of tests run: 1

Test 6.8.3.5-14 shows that the otherwise clause in a case statement has not been implemented.

Pascal Validation Suite Report

Pascal processor identification

This Pascal-VU compiler produces code for an EM-1 machine as defined in [1]. It is up to the implementor of the EM-1 machine whether errors like integer overflow, undefined operand and range bound error are recognized or not. Therefore it depends on the EM-1 machine implementation whether these errors are recognized in Pascal programs or not. The validation suite results of all known implementations are given.

There does not (yet) exist a hardware EM-1 machine. Therefore, EM-1 programs must be interpreted, or translated into instructions for a target machine. The following implementations currently exist:

Implementation 1: an interpreter running on a PDP-11 (using UNIX). The normal mode of operation for this interpreter is to check for undefined integers, overflow, range errors etc.

Implementation 2: a translator into PDP-11 instructions (using UNIX). Less checks are performed than in the interpreter, because the translator is intended to speed up the execution of well-debugged programs.

Test 6.6.3.3-3:
Test 6.8.2.2-2:
Several pointer type definitions (^rekord) referring to the same record type are incompatible.

Test 6.6.3.4-2:
Only a single procedure identifier is allowed in a formal procedure parameter section.

Test 6.9.4-4:
Reals printed in scientific notation always contains an exponent part, even for exponent equal to zero.

Latest standard proposal

A newer version of the proposal is received in November 1979. Because of the differences between these versions the following tests are changed:

Test 6.1.5-6:
The case of any letter occurring anywhere outside of a character-string shall be insignificant in that occurrence to the meaning of the program.

Test 6.4.3.3-3:
Test 6.4.3.3-11:
Test 6.4.3.3-12:
Definition of an empty record is not allowed.

Test 6.4.3.3-10:
The case-constants introducing the variants shall be of ordinal-type that is compatible with the tag-type.

Test 6.5.1-1:
The type of the component of a file-type shall be neither a file-type nor a structured-type with a file component.

Test 6.9.4-4:
Test 6.9.4-5:
The character indicating the exponent part of a real as written in scientific notation is either 'e' or 'E'.

Test 6.9.4-4:
The representation of a positive real in fixed point format does not include a space if it does not fit in the specified field width.

Test 6.9.4-7:
The case of each of the characters written as representation for a Boolean is implementation-defined.

Test 6.9.4-9:
Zero or negative field width is allowed in write-parameters.

Conformance tests

Number of tests passed = 138
Number of tests failed = 1

Details of failed tests

Test 6.1.2-3:
Character sequences starting with the 8 characters 'procedur' or 'function' are erroneously classified as the word-symbols 'procedure' and 'function'.

Test Conditions

Tester: J.W.Stevenson
Date: December 19, 1979
Validation Suite version: 2.0, dated June 19, 1979

The final test run is made with a slightly modified validation suite. The changes made can be divided into the following categories:

Typing errors

Test 6.4.3.5-1:
the identifier 'ptrtoi' must be a type-identifier, not a variable-identifier.

Test 6.6.3.3-1:
The type of 'colone' should probably be 'subrange', not 'colour', because the types of actual and formal variable parameters should be identical.

Test 6.6.3.1-5:
In passing a procedure as actual parameter the parameters must not be specified. So line 29 must be changed to
conform(alsoconforms)

Test 6.6.5.3-1:
This test is incorrectly terminated by 'END.' instead of 'end.'.

Test 6.6.1-7:
The terminating 'end.' is incorrectly preceded by a space.

Test 6.9.4-14:
The program parameter 'f' must be removed.

Portability problems

Test 6.6.3.1-2:
A set of integer subrange containing more than 16 elements may give problems for some implementations. A special option must be provided to the Pascal-VU compiler, specifying the number of elements.

Test 6.6.6.2-3:
Not all implementations support reals with 9 decimals of precision. The precision supported by Pascal-VU is about 7 decimals (24 bits).

Erroneous programs

Some tests did not conform to the standard proposal of February 1979. It is this version of the standard proposal that is used by the authors of the validation suite.

Test 6.3-1:
Test 6.6.3.1-4:
Test 6.4.5-5:
The meaning of these test program is altered by the truncation of their identifiers to eight characters.

Test 6.4.3.3-1:
A record definition consisting of a single semicolon is illegal.

Deviance tests

Number of deviations correctly detected = 81
Number of tests not detecting deviations = 12

Details of deviations

The following tests fail because the Pascal-VU compiler only generates a warning that does not prevent to run the tests.

Test 6.2.1-5:
A declared label that is never defined produces a warning.

Test 6.6.2-5:
A warning is produced if there is no assignment to a function-identifier.

The following tests are compiled without any errors while they do not conform to the standard.

Test 6.2.2-4:
Test 6.3-6:
Test 6.4.1-3:
Undetected scope error. The scope of an identifier should start at the beginning of the block in which it is declared. In the Pascal-VU compiler the scope starts just after the declaration, however.

Test 6.8.2.4-2:
Test 6.8.2.4-3:
Test 6.8.2.4-4:
The Pascal-VU compiler does not restrict the places from where you may jump to a label by a goto-statement.

Test 6.8.3.9-2:
Test 6.8.3.9-3:
Test 6.8.3.9-4:
Test 6.8.3.9-16:
There are no errors produced for assignments to a variable in use as control-variable of a for-statement.

Error handling

The results depend on the EM-1 implementation.

Number of errors correctly detected =
Implementation 1: 26
Implementation 2: 12
Number of errors not detected =
Implementation 1: 19
Implementation 2: 33

Details of errors not detected

Test 6.2.1-7:
It is allowed to print all integer values, even the special 'undefined' value.

Test 6.4.3.3-5:
Test 6.4.3.3-6:
Test 6.4.3.3-7:
Test 6.4.3.3-8:
The notion of 'current variant' is not implemented, not even if a tagfield is present.

Test 6.4.6-4:
Test 6.4.6-5:
Implementation 2: Subrange bounds are not checked.

Test 6.4.6-7:
Test 6.4.6-8:
Test 6.7.2.4-2:
If the base-type of a set is a subrange, then the set elements are not checked against the bounds of the subrange. Only the host-type of this subrange-type is relevant for Pascal-VU.

Test 6.5.3.2-1:
Implementation 2: Array bounds are not checked.

Test 6.5.4-1:
Test 6.5.4-2:
Implementation 2: Nil or undefined pointers are not detected.

Test 6.6.2-6:
An undefined function result is not detected, because it is never used in an expression.

Test 6.6.5.2-6:
Test 6.6.5.2-7:
Changing the file position while the window is in use as actual variable parameter or as an element of the record variable list of a with-statement is not detected.

Test 6.6.5.3-3:
Test 6.6.5.3-4:
Implementation 2: Disposing nil or an undefined pointer is not detected.

Test 6.6.5.3-5:
Test 6.6.5.3-6:
Disposing a variable while it is in use as actual variable parameter or as an element of the record variable list of a with-statement is not detected.

Test 6.6.5.3-7:
Test 6.6.5.3-8:
Test 6.6.5.3-9:
It is not detected that a record variable, created with the variant form of new, is used as an operand in an expression or as the variable in an assignment or as an actual value parameter.

Test 6.6.6.4-4:
Test 6.6.6.4-5:
Test 6.6.6.4-7:
Implementation 2: There are no range checks for pred, succ and chr.

Test 6.7.2.2-3:
Test 6.7.2.2-6:
Test 6.7.2.2-7:
Test 6.7.2.2-8:
Implementation 2: Division by 0 or integer overflow is not detected.

Implementation dependence

Number of test run = 15
Number of tests incorrectly handled = 0

Details of implementation dependence

Test 6.4.2.2-7:
Maxint = 32767

Test 6.4.3.4-2:
'set of char' allowed.

Test 6.4.3.4-4:
Up to 256 elements in the range 0..255 in a set.

Test 6.6.6.1-1:
Standard procedures and functions are not allowed as parameter.

Test 6.6.6.2-11:
Details of the machine characteristics regarding real numbers.

Test 6.7.2.3-2:
Test 6.7.2.3-3:
Boolean expressions fully evaluated.

Test 6.8.2.2-1:
Test 6.8.2.2-2:
The expression in an assignment statement is evaluated before the variable selection if this involves pointer dereferencing or array indexing.

Test 6.9.4-5:
Number of digits for the exponent is 2.

Test 6.9.4.11:
The default field widths for integer, Boolean and real are 6, 5 and 13.

Test 6.10-2:
Rewrite(output) is a no-op.

Test 6.11-1:
Test 6.11-2:
Test 6.11-3:
Alternate comment delimiters implemented, but not the other equivalent symbols.

Quality measurement

Number of tests run = 23
Number of tests incorrectly handled = 0

Results of tests

Test 5.2.2-1:
Test 6.1.3-3:
Only 8 characters are significant in identifiers.

Test 6.1.8-4:
Both ';' and '{' cause a warning message if they are found inside comments.

Test 6.2.1-8:
Test 6.2.1-9:
Test 6.5.1-2:
Large lists of declarations are possible in each block.

Test 6.4.3.2-4:
An 'array[integer] of' is not allowed.

Test 6.4.3.3-9:
Variant fields of a record occupy the same space, using the declared order.

Test 6.4.3.4-5:
Size and speed of Warshall's algorithm depends on the implementation of EM-1

Implementation 1:
size: 81 bytes
speed: 4.20 seconds

Implementation 2:
size: 204 bytes
speed: 0.62 seconds

Test 6.6.1.7:
At least 15 levels of nested procedures allowed.

Test 6.7.2.2-4:
'div' is correctly implemented for negative operands.

Test 6.8.3.5-2:
The compiler requires case constants to be compatible with the case selector.

Test 6.8.3.5-8:
Large case statements are possible.

Test 6.8.3.9-18:
The value of the control variable of a normally terminated for-statement is equal to the final value.

Test 6.8.3.9-20:
At least 20 nested for-statements allowed.

Test 6.8.3.10-7:
At least 15 nested with-statements allowed.

Test 6.9.4-10:
Line marker appended at end of job if the last character written is not a line marker.

Test 6.9.4-14:
Recursive i/o using the same file allowed.

The following 5 tests test the mathematical functions. For each the following three quality measures are extracted from the test results:

meanRE: mean relative error.
maxRE: maximum relative error
rmsRE: root-mean-square relative error

```
Test 6.6.6.2-6:
     Test sqrt(x): no errors and correct results.

Test 6.6.6.2-7:
     Test arctan(x): may cause underflow or underflow errors.
          meanRE: 2 ** -30.46
          maxRE:  2 ** -22.80
          rmsRE:  2 ** -24.33

Test 6.6.6.2-8:
     Test exp(x): may cause underflow or overflow errors.
          meanRE: 2 ** -25.37
          maxRE:  2 ** -17.62
          rmsRE:  2 ** -19.56

Test 6.6.6.2-9:
     Test sin(x): may cause underflow errors.
          meanRE: 2 ** -22.98
          maxRE:  2 ** -10.43
          rmsRE:  2 ** -15.59

     Test cos(x): may cause underflow errors.
          meanRE: 2 ** -21.69
          maxRE:  2 ** - 8.28
          rmsRE:  2 ** -13.37

Test 6.6.6.2-10:
     Test ln(x): no errors
          meanRE: 2 ** -25.12
          maxRE:  2 ** -21.97
          rmsRE:  2 ** -23.75
```

Extensions

```
Number of test run = 0
```

References

[1] A.S.Tanenbaum, J.W.Stevenson, J.M.van Staveren, "Description of an experimental machine architecture for use of block structured languages", Informatica rapport IR-54.

[2] ISO standard proposal ISO/TC97/SC5-N462, dated February 1979. The same proposal, in slightly modified form, can be found in: A.M.Addyman e.a., "A draft description of Pascal", Software, practice and experience, May 1979. An improved version, received November 1979, is followed as much as possible for the current Pascal-VU.

☆ ☆ ☆ ☆ ☆ ☆ ☆ ☆

CDC-6000 RELEASE 3

PASCAL VALIDATION SUITE REPORT

Pascal Processor Identification

Computer: Control Data Corp. CYBER 74, running NOS 1.3

Processor: CDC-6000 Release 3 (Zurich Compiler) of January, 1979

Test Conditions

Tester: Rick L. Marcus

Date: January, 1980

Validation Suite Version: 2.2

Conformance Tests

Number of tests passed: 128

Number of tests failed: 11

Details of failed tests

Test 6.1.8-3 is not relevant; only one form of comment is allowed.

Test 6.2.2-3 fails because the compiler thinks that the scope of node = real covers procedure ouch.

Test 6.2.2-8 fails because assignment to a function is allowed only within the function body.

Test 6.4.3.3-1 fails because the declaration for an empty record (D) is not allowed. If the semi-colon is removed from the record definition then there is no error, which can be seen in the next test, 6.4.3.3-3.

Test 6.4.3.3-4 fails because the tag-field in a record may not redefine an existing type declared elsewhere.

Test 6.5.1-1 fails because the compiler does not allow a file of record... where the record contains a file as a field. I believe the latest version of the standard changes this. Our compiler will pass the test if files of files are not allowed.

Test 6.6.3.1-1 fails in procedure Testtwo because of 'strict' type checking. Passing a variable of type colour as a parameter of type subrange causes the error. Passing as a value paramter is allowed(i.e., procedure Testone passes the test).

Test 6.8.3.5-4 fails because the range of case labels is too large.

Test 6.9.1-1 fails because eoln is not necessarily true after the last character written on a line. The operating system pads to an even number of characters on a line with blanks.

Test 6.9.4-4 fails because the test assumes only two places in the exponent field while there are three on our CDC systems.

Test 6.9.4-7 fails because Booleans are right justified on CDC 6000 Pascal, not left as in the test. I believe the latest standard assumes right justification, so that the compiler would pass the test in that case.

Deviance Tests

Number of deviations correctly detected: 76

Number of tests not detecting erroneous deviations: 18

Details of deviations

Test 6.1.2-1 shows that nil is not a reserved word.

Tests 6.1.5-6 is not relevant as only upper case is allowed anywhere in a Pascal program.

Test 6.2.1-5 shows that a label may be declared without being used anywhere in a program.

Tests 6.2.2-4, 6.3-6, 6.4.1-3 show that a common scope error was not detected by the compiler.

Test 6.6.2-5 shows that a function need not be assigned a value inside its body. The value of A after the assignment (A := ILLEGAL(A)) is zero.

Test 6.6.3.5-2 shows that strict type compatibility of functions passed as parameters is not required.

Tests 6.8.2.4-2, 6.8.2.4-3, 6.8.2.4-4 show that a goto between branches of a statement is permitted.

Tests 6.8.3.9-2, 6.8.3.9-3, 6.8.3.9-4, and 6.8.3.9-16 show that an assignment may be made to a for statement control variable.

Test 6.8.3.9-14 shows that a for loop control variable may be a variable global to the whole program.

Test 6.8.3.9-19 shows that in nested for loops, if both have the same control variable, then the value gets changed by the inner loop and falls out of the outer loop after 1 iteration.

Test 6.9.4-9 shows that characters may be written even if the field width is too small.

Error Handling

Number of errors correctly detected: 24

Number of errors not detected: 21

Number of tests incorrectly handled: 1

Details of errors not detected

Test 6.2.1-7 shows that the value of I is that which is left over from procedure q (I=3).

Tests 6.4.3.3-5/6/7/8 indicate that no checking is done on the tag field of variant records.

Test 6.4.3.3-12 shows that an empty record can be assigned an undefined empty value.

Test 6.4.6-8 shows that strict type compatibility is not enforced for sets passed by value.

Test 6.6.2-6: The error was not detected. The value of the variable CIRCLEAREA was zero after the assignment. It seems that a function is assigned the value zero if no assignment is made in its body.

Test 6.6.5.2-2 fails to catch the error because of system padding of blanks to an even number of blanks.

Test 6.6.5.2-6/7 shows that I/O is not implemented according to the standard.

Test 6.6.5.3-5 fails because no check is made by the runtime system to see if the variable being disposed of is a parameter to the procedure which calls dispose.

Tests 6.6.5.3-6/7/8/9 all fail.

Tests 6.7.2.2-6/7 fail because an integer variable does not cause an overflow error when it is over the value of maxint.

Tests 6.8.3.9-5/6 show that the value of an integer control variable is set to -576460752303423487 after the for loop.

Test 6.8.3.9-17 shows that two nested for loops may have the same control variable.

Details of tests incorrectly handled

Test 6.6.6.3-3: An overflow of the real variable reel caused termination of the program.

Implementationdefined

Number of tests run: 15

Number of tests incorrectly handled: 0

Deatails of implementation-dependence

Test 6.4.2.2-7 shows maxint to be 281474976710655.

Tests 6.4.3.4-2/4 show that set bounds must be positive, have no element whose ordinal is greater than 58, and that set of char is not legal.

Test 6.6.6.1-1 indicates that standard procedures and functions are not allowed to be passed as parameters to procedures and functions.

Test 6.6.6.2-11 details some machine characteristics regarding number formats.

Tests 6.7.2.3.2/3 show Boolean expressions are fully evaluated.

Tests 6.8.2.2-1/2 show that a variable is selected before an expression is evaluated in an assignment statement.

Test 6.9.4-5 shows that the number of digits in an exponent is 3.

Test 6.9.4-11 details the default field width specifications: 10 for integers and Booleans and 22 for reals.

Test 6.10-2 shows that a rewrite is allowed on the file output, but that it has no effect (i.e., output is not rewritten) unless there is an actual local file of a different name which replaces output on the control statement to execute the program.

Test 6.11-1/2/3 show that alternate comment delimiters and other alternate symbols have not been implemented.

Quality

Number of tests run: 23

Number of tests incorrectly handled: 0

Results of quality measurements

Tests 5.2.2-1 and 6.1.3-3 show that identifiers are not distinguished over their whole length; only the first 10 characters are significant.

Test 6.1.8-4 shows that no warning is given if a valid statement or semicolon is embedded in a comment.

Tests 6.2.1-8/9 and 6.5.1-2 indicate that large lists of declarations may be made in each block.

Test 6.4.3.2-4 shows that an array with an indextype of INTEGER is not permitted. At this site the use of INTEGER for an indextype is permitted only in the current implementation of dynamic arrays.

Test 6.4.3.3-9 shows that the variant fields of a record occupy the same space, using the declared order.

Test 6.4.3.4-5 (Warshall's algorithm) took 0.236 seconds CPU time and 171 words (10260 bits) on the CDC CYBER 74.

Test 6.6.1-7 shows that procedures cannot be nested to a level greater than 9.

Tests 6.6.6.2-6/7/8/9/10 tested the sqrt, atan, exp, sin/cos, and ln functions and all tests showed there were no significant errors in their values.

Test 6.7.2.2-4 shows that div and mod have been implemented consistently. mod returns the remainder of div.

Test 6.8.3.5-2 shows that case constants do not have to be of the same type as the case index, if the case index is a subrange, but the constants must be compatible with the case index.

Test 6.8.3.5-8 shows that a large case statement is permissible (>256 selections).

Test 6.8.3.9-18 shows that the use of a control variable is allowed after the for loop. The run-time system catches the use of the control variable this time because after exiting the loop the variable is set to the value found in Test 6.8.3.5.9-5, and the case variable is out of range.

Tests 6.8.3.9-20 and 6.8.3.10-7 indicate that for and with statements may be nested to a depth greater than 15.

Test 6.9.4-10 shows that file buffers are flushed at the end of a the program.

Test 6.9.4-14 indicates that recursive I/O is permitted, using the same file.

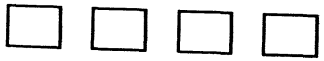
Extension

Number of tests run: 1

Number of tests incorrectly handled: 0

Details of extensions

Test 6.8.3.5-14 shows that the 'OTHERWISE' clause has been implemented in a case statement.



TI PASCAL

0. DATE/VERSION

Release 1.6.0, January 1980.

1. IMPLEMENTOR/MAINTAINER/DISTRIBUTER

Implemented by Texas Instruments. Information is available from TI sales offices, or write to

Texas Instruments
Digital Systems Group, MS784
P. O. Box 1444
Houston, Texas 77001

or call (512) 250-7305. Problems should be reported to

Texas Instruments
Software Sustaining, MS2188
P. O. Box 2909
Austin, Texas 78769

or call (512) 250-7407.

2. MACHINE

The compiler runs on a TI 990/10 or 990/12. The compiled object code can be linked for execution on any member of the 990 computer family.

3. SYSTEM CONFIGURATION

The compiler runs under the DX10 operating system (release 3) and requires at least a TI DS990 Model 4 system, which includes a 990/10 with 128K bytes of memory and a 10 megabyte disk. (More than 128K of memory may be required, depending on the size of the operating system.) Compiled programs can be executed on any FS990 or DS990 system, using the TX5, TX990, or DX10 operating systems.

4. DISTRIBUTION

Available on magnetic tape or disk pack. Contact a TI salesman for a price quotation and further details.

5. DOCUMENTATION

Complete user-level documentation is given in the "TI Pascal User's Manual", TI part number 946290-9701.

6. MAINTENANCE POLICY

TI Pascal is a fully supported product. Bug reports are welcomed and maintainence and further development work are in progress.

7. STANDARD

TI Pascal conforms to "standard" Pascal, with the following principal exceptions:

* A GOTO cannot be used to jump out of a procedure.
* The control variable of a FOR statement is local to the loop.
* The precedance of Boolean operators has been modified to be the same as in Algol and Fortran.
* The standard procedures GET and PUT have been replaced by generalized READ and WRITE procedures.

TI Pascal has a number of extensions to standard Pascal, including random access files, dynamic arrays, ESCAPE and ASSERT statements, optional OTHERWISE clause on CASE statements, and formatted READ.

8. MEASUREMENTS

The compiler occupies a 64K byte memory region. Compilation speeds are comparable to the 990 Fortran compiler.

9. RELIABILITY

The system has been used by several different groups within TI since October of 1977, and by a number of outside customers since May of 1978. Updates have been released in January 1979 and January 1980. This long history of extensive use and maintainance makes this a reasonably stable and reliable product.

10. DEVELOPMENT METHOD

The compiler produces object code which is link-edited with run-time support routines to form a directly executable program. The compiler is written in Pascal and is self-compiling.

11. LIBRARY SUPPORT

TI Pascal supports separate compilation of routines and allows linking with routines written in Fortran or assembly language.

Intel 8080/8085 Meta Tech

Specializing In Innovative Information Processing

Pascal/MT Implementation Specification

-0- Date: November 8, 1979
Version: Release 2.5

-1- Distributed, Implemented and Maintained by:

MetaTech, 8672-I Via Mallorca, La Jolla, Ca. 92037
(714) 223-5566 x289 or (714) 455-6618

-2- Machine: Intel 8080/8085 and Zilog Z80

-3- System Configuration:
Pascal/MT operates under the CP/M operating system (or and equivalent system such as CDOS, IMDOS, etc.) in a minimum of 32K bytes of memory.

The package consists of a compiler and symbolic debugger and generates 8080/Z80 object code directly from the Pascal program source.

The symbolic debugging package is optionally copied into the output object file by the compiler.

-4- Distribution:
The Pascal/MT package is distributed on a single density 8-inch floppy diskette which contains:

The compiler for Pascal/MT
The symbolic debugging package
The text for the compiler error messages
Two utility programs written in Pascal/MT to illustrate the facilities of the language

Cost of a single system license for Pascal/MT (includes manual) is $99.95
Manual available for $30.00
Source for the run-time package is $50.00

Master Charge, Visa, UPS COD, and Purchase Orders

-5- Standard:
Pascal/MT implements (in 2.5) a subset of the full Pascal language. This was done to generate both space and time efficient code for 8-bit microcomputers.

Pascal/MT also contains a number of "built-in" procedures. This allows source code using these procedures to be portable to other systems providing appropriate routines are implemented on the other systems.

Pascal/MT omits the following features from the Pascal standard (Jensen & Wirth 2nd Ed.):
*No LABEL declaration and therefore no GOTOs
Non-standard file support for CP/M files
*Enumeration and Record types not implemented
PACKED is ignored on boolean arrays
All variables and parameters are allocated statically
Items marked with a * are being implemented in the subsequent releases of Pascal/MT.

-6- Extensions:
Pascal/MT contains the following extensions (in release 2.5):
Pre-declared arrays "INPUT" and "OUTPUT" for manipulating I/O ports directly.
EXTERNAL assembly language procedure declarations for using pre-assembled routines (using PL/M parameter passing)
OPEN, CLOSE, DELETE, CREATE, BLOCKREAD, BLOCKWRITE routines for accessing CP/M files.
Logical un-typed boolean operators for and (&) or (!) and not (~)
18-digit BCD arithmetic package.

-7- Measurements:
Compilation speed is approximately 600 lines/min. 6K bytes symbol table space is available in a 32K system and 38K bytes table space is available in a 64K system. Run-time code (without debugger) is 5 to 10 times faster than P-code systems, and is 1.5 to 3 times larger than P-code systems (but Pascal/MT requires no interpreter).

-8- Availability:
Pascal/MT Release 2.5 is available immediately.

Enhanced releases will be made periodically throughout the next year.

Also available from: FMG Corp (817) 294-2510 for TRS-80
Lifeboat Assoc. (212) 580-0082 for all formats

IBM 370-165 (Weizmann)

(See GOLEM B (Weizmann))

Motorola 6800 (Dynasoft Systems)

This is a compiler for a cassette-based system, and sells for $35.

Dynasoft Systems
POB 51
Windsor Junction, North Saskatchewan B0N 2V0
Canada

Motorola 6809 (Motorola)

MOTOROLA 6809 PASCAL - CHECKLIST FOR PASCAL NEWS

0. DATE/VERSION

12 December 1979
Version 1.0 released September 1979
Version 1.1 to be released February 1980

1. IMPLEMENTOR/DISTRIBUTOR/MAINTAINER

Motorola Microsystems
P.O. Box 20906
Phoenix, Arizona 85036
(602) 831-4108

2. MACHINE

Motorola 6809 EXORciser

3. SYSTEM CONFIGURATION

MDOS09 03.00 running on 6809 EXORciser with 56K bytes and floppy-disk drive.

4. DISTRIBUTION

On floppy diskette (M6809PASCLI) for $1500 from
Motorola Microsystems
P.O. Box 20906
Phoenix, Arizona 85036
(602) 962-3226

Orders should be placed through local Motorola Sales Office or Distributor

5. DOCUMENTATION

Motorola Pascal Language Manual (M68PLM(D1)) describing Motorola implementation (56 pages). 6809 Pascal Interpreter User's Guide (M68Ø9PASCLI(D1)) describing operation of interpreter (48 pages).

6. MAINTENANCE POLICY

Bugs should be reported to software support. Subsequent releases will include corrections.

7. STANDARD

Restrictions: May not specify formal parameters which are procedure or function identifiers. Floating point numbers are not implemented. Packed attribute has no effect. All will be implemented in future releases. Enhancements: Address specification for variables; alpharumeric labels; an exit statement; external procedure and function declarations; non-decimal integers; otherwise clause in case statement; runtime file assignments; structured function values; string variables and string functions.

8. MEASUREMENTS

Compiles in 56K bytes. Runtime support requires 3-4K byte interpreter module.

9. RELIABILITY

Very good--first released in September 1979 with few major problems reported.

10. DEVELOPMENT METHOD

One pass recursive descent compiler generates variable length P-code. One pass P-assembler (second release) generates a compact, position-independent code for interpreter. Code and interpreter both ROMable for use in non-EXORciser environment.

11. LIBRARY SUPPORT

Standard Pascal procedures and functions, plus the ability to link assembly language routines.

RCA 1802 (Golden River)

The information on this compiler is unclear. It appears to be all or partly in ROM, and sells for £ 40.

The Golden River Co., Ltd.
Telford Road.
Bicester, Oxfordshire OX6 ØUL
England

Zilog Z-80 (Digital Marketing)

(See Zilog Z-80 (Digital Marketing))

Zilog Z-80 (Ithaca Intersystems)

This compiler runs under CP/M and produces macro-assembler code. The price is $275.

Ithaca Intersystems (formerly Ithaca Audio)
1650 Hanshaw Road
P.O. Box 91
Ithaca, NY 14850

Zilog Z-80 (MetaTech)

(See Zilog Z-80 (MetaTech))

GOLEM B (Weizmann)

מכון ויצמן למדע
THE WEIZMANN INSTITUTE OF SCIENCE
REHOVOT · ISRAEL רחובות · ישראל

DEPARTMENT OF APPLIED MATHEMATICS המחלקה למתמטיקה שמושית

Pascal User's Group
C/o J. Miner
University Computer Center: 227 EX
208 SE Union St.
University of Minnesota
Minneapolis, MN 55455

September 5, 1979

Dear Mr. Miner,

I have transported the Zurich P4 Compiler to the GOLEM B computer of the Weizmann Institute. Following is a checklist for Implementation Notes:

0. Date/Version. 79/09/03

1. Distributor/Implementor/Maintainer:

   W. Silverman
   c/o Dept. of Applied Mathematics
   The Weizmann Institute of Science
   Rehovot, Israel.

2. Machine: GOLEM B, 370-165.

3. System Configuration: GOBOS for GOLEM B (designed and built by WI). Also produces P-CODE on our 370-165. Variants produce P-CODE for the GA-16 and the Z80; a loader, written in PASCAL is available for the latter, and an interpreter is being checked out on our Z80 simulator and on the TEKTRONIX 8001/8002A µProcessor Lab.

4. Distribution: Source of compiler, configured for your machine as is P4, with a few additional parameters, and of our Loader and additional package as they become available, on magnetic tape (9-track, 1600 BPI or 7-track as required) within Israel. Send mini-tape to distributor - mailing costs only. Special arrangements possible outside Israel.

5. Documentation: Same as P4 system plus additional P-Code and extra parameters descriptions.

6. Maintenance Policy: Bug reports receive prompt attention and replies. Various optimization programs will be announced as available.

7. Standard:

   Extensions to P4 (Standard):
   Multiple global text files permitted and "FILE OF CHAR" properly processed;
   Procedure/Function may be declared as formal parameter (no run-time check for argument match);

   PACK,UNPACK,ROUND,REWRITE,RESET implemented;

   e:el:e2 implemented for real e in WRITE-list;

   MAXINT accessible as standard constant.

   Non-Standard Extensions:

   FORTRAN, EXTERN and independent compilation option (*$E+*);

   Additional digraphs and operator codes (e.g. "(.",".)","&").

8. Measurements:

   - Compilation speed: 1300 characters/second (measured compiling itself; 4442 lines x 80 characters per line in 280 seconds - 300 seconds with listing).
   - Compilation space: 288000 8-bit bytes (this can be reduced somewhat from the actual $11B64_{16}$ 4-byte words of storage, by reducing the stack/heap which is nominally 128K bytes for GOLEM B - the basic level-0 stack requirement is 6700 bytes, plus 700 bytes per recursion level of BODY and a basic procedural overhead of 32 bytes per nested call).
   - Execution speed: Approximately 1/5 as fast as PASCAL 8000 on our 370/165 (the GOLEM is intrinsically 1/2 as fast).
   - Execution space: 8.3 bytes / P-Code instruction (peep-hole optimization improves this figure dramatically), plus data storage as follows:

Item	Size	Allignment
Stack element	8 bytes	8 bytes
Real	8 "	8 "
Integer	4 "	4 "
Pointer	4 "	4 "
Character	1 byte	byte
Set	1-8 bytes	byte
Boolean	1 bit	bit

Note that Boolean arrays are optimally stored, 1-bit/element; the cost in access overhead is modest. Declared scalars are represented as integers.

CABLE ADDRESS: WEIZINST (Israel) מען למברקים: PHONE: (054)82111-83111 טלפון: TELEX: 31934 טלקס:

9. Reliability:

Excellent - this is primarily due to the high quality of the P4 system we received from Zurich. We have conserved their design and implementation principles in all modifications. Two sites, both of them at the WI, are currently using the system. Two others are considering it.

10. Development Methods

The P4 Compiler was transported to the GOLEM in approximately three months (real time) by one person. The P-Code is expanded as macros by our Assembler to produce a mixture of in-line instructions and subroutine calls. The Assembly-language system consists of 3430 source lines, including all macro definitions and full run-time support. Total effort to date is approximately 6 man-months by professional programmers plus 2 student-months. Included in this effort are 850 lines of modifications and extensions to P4, written in PASCAL, replacing and extending 300 lines of the received compiler.

The implementor had previously been project manager for CDC FORTRAN '63, supervised the development of several other compilers and written numerous Cross-Asemblers and Simulators; although familiar with ALGOL he had no previous knowledge of PASCAL.

11. The FORTRAN and EXTERN extensions permit access to FORTRAN libraries (specifically NAG and IMSL) and to independantly compiled PASCAL procedures and functions. All linkage is via our system linking loader, so normal JCL suffices:

e.g. a) PASCAL (SOURCE)
EXAMPLE

compiles and executes the program EXAMPLE from the file SOURCE.

b) PASCAL (S1)
PASCAL (S2,BINARY1)
PASCAL (S3,BINARY2)
FORTRAN (S4,BINARY3)
LOADNAG,EXAMPLE
EXAMPLE

compiles 4 files and links their programs with binary and NAG routines and executes the resultant load-module, EXAMPLE, which has the same name (in this case) as the PROGRAM statement of the principal file.

No automatic text copying is supplied, however, various editors obviate the need for this facility with COPY,DECK,COMDECK,COMMON and INCLUDE commands.

A symbolic-dump-table option produces a symbolic file which is used by a PASCAL-coded Post-Mortem dump procedure to produce a symbolic dump (one new P-CODE instruction is produced for this purpose).

We have two Master's thesis projects developing transportable optimization programs for global and peep-hole optimization. I'll keep you informed of their progress. We're also developing a Cross-compilation support system to provide PASCAL capability to all the lab-computers (Mini's and Micro's) connected to our major computer complex.

Sincerely,

William Silverman

P.S. דרישת שלום to Kris and Elaine Frankowski and hello to Larry Liddiard, et al in the Computer Science Dept.

Intel 8085/8080 (Digital Marketing)

This compiler runs under CP/M and is a Pascal-P descendant. The price is $350.

Digital Marketing
2670 Cherry Lane
Walnut Creek, CA 994596

GAMMA TECHNOLOGY Pascal for Data General AOS Systems

Implementation Checklist

0) DATE/VERSION:
AOS Pascal Revision 2 - September 1979
Checklist date: November 16, 1979

1) DISTRIBUTOR:
Gamma Technology, Inc.
2452 Embarcadero Way
Palo Alto, CA 94303
(415) 856-7421
TWX: 910-373-1296

IMPLEMENTOR/MAINTAINER:
Pete Goodeve
3012 Deakin Street
Berkeley, CA 94705

2) MACHINE/SYSTEM CONFIGURATION:
Data General Eclipse and M600 Series

3) SYSTEM CONFIGURATION:
AOS Rev 2.00 or later
Floating point hardware

4) DISTRIBUTION: $500 package comprises a 9-track, 800 bpi magnetic tape and documentation (package price is $50 if it is an upgrade to a previously purchased release-1 system.
Tape is in AOS dump format, containing a complete system, documentation, and sources. Included at no charge are some public domain Pascal utilities based on those supplied by the University of Minnesota.

5) DOCUMENTATION: An instruction manual gives details of usage under AOS; a current textbook should be referred to for knowledge of the Pascal language itself.
Differences from the (draft) standard (and from previous versions) are described in reference sections of the manual.
An AOS "HELP" file is supplied, and also documentation for the utilities.
All documentation is also in machine-retrievable form.
An up-to-date list of bugs and notes will be maintained.

6) MAINTENANCE POLICY: Bug reports (in writing) are encouraged; please send them to the distributor (Gamma Technology). The system is expected to be stable; no incremental upgrades are planned, but fixes will be distributed. Any future major development will depend on demand.

7) STANDARD: The compiler is a considerably enhanced derivative of P-4 (christened "P-5") with many major restrictions of the original removed.

Restrictions:
PACKED is ignored; PACK and UNPACK are not implemented.
DISPOSE is not implemented; heap management is by MARK and RELEASE.
Parameters may not be Procedural or Functional.
Subrange set constructors are not recognized.
There are restrictions on READ and WRITE (but not on GET and PUT) for files either passed as parameters or coded as non-text.
Only four text files may be in use at one time (no restriction on other types).

Enhancements over earlier versions (P-4):
Files may be of any type (except FILE).
Any (global level) files may be specified external in program header.
Full ASCII is supported; lower and upper case alphabetics are equivalent identifiers. Braces may be used as comment delimiters.
String constants may be up to one line in length.
Format control of real output is as defined by the standard.
TEXT and MAXINT are predeclared.
GOTOs must be to a line within enclosing scope (standard Pascal).
Stack frame allocation is improved.

Language enhancements:
External procedures (Pascal or assembly code).
HALT (or HALT(n)) abnormal termination feature.
Random access to all files.

AOS features:
Compile command options (selecting e.g. cross-reference listing, binary only, syntax check only, etc.).
Max stack/heap space allocated can be specified at both compile and execution time.
External files may be specified in execution command.

8) MEASUREMENTS: No real timing tests have yet been made, but compiler compiles itself (on a quiet system) in 10 or 11 minutes.
The run-time interpreter occupies about 9 Kbytes. In addition to this, and the space needed for the program's P-code, a default of 4K bytes is allocated for run-time stack and heap space; this can be increased or decreased by the user at compile and/or run time -- the range is from 2K bytes up to the limits of the machine.

9) RELIABILITY: Excellent, over the two months it has been running at the development site. No Pascal program has yet managed to cause a system crash (unlike other languages runing under AOS).
Revision 1 is now in use at abut 20 sites, with a good reliability record. A few slight problems (mainly with stack overflow) found in that revision have been fixed in the new one.

10) DEVELOPMENT METHOD: This is a fast P-code interpreter system. The compiler generates an extended, machine-independent symbolic P-code, which is then translated and assembled into a compact binary form; this is bound with the interpreter to create an executable program file. The sequence from source to program file is managed automatically by a single user command.
This "P-5" compiler has been developed directly from the Lancaster version of P-4. It should be completely transportable, except that it assumes the character set is ASCII. Aside from its use of HALT(n), the compiler is written entirely in standard Pascal. It was necessary, however, to split the compiler P-code into overlay segments, so that large programs can be compiled (the overlay scheme is not available to user programs).
The P-code translator is written in Pascal, and the run-time system in Eclipse assembly language.

11) LIBRARY SUPPORT: Pascal cannot be linked to other languages for the Eclipse (except assembly language), because each has its own stack format.
External procedure modules may be compiled separately and linked to a main (Pascal) program. External procedures may also be written in assembly language.
One or two library procedures are supplied with the system (for example, for extra file management functions), but no general library is envisaged aside from the utilities already supplied.

IMPLEMENTATION NOTES ONE PURPOSE COUPON

0. **DATE**

1. **IMPLEMENTOR/MAINTAINER/DISTRIBUTOR** *(* Give a person, address and phone number. *)*

2. **MACHINE/SYSTEM CONFIGURATION** *(* Any known limits on the configuration or support software required, e.g. operating system. *)*

3. **DISTRIBUTION** *(* Who to ask, how it comes, in what options, and at what price. *)*

4. **DOCUMENTATION** *(* What is available and where. *)*

5. **MAINTENANCE** *(* Is it unmaintained, fully maintained, etc? *)*

6. **STANDARD** *(* How does it measure up to standard Pascal? Is it a subset? Extended? How.*)*

7. **MEASUREMENTS** *(* Of its speed or space. *)*

8. **RELIABILITY** *(* Any information about field use or sites installed. *)*

9. **DEVELOPMENT METHOD** *(* How was it developed and what was it written in? *)*

10. **LIBRARY SUPPORT** *(* Any other support for compiler in the form of linkages to other languages, source libraries, etc. *)*

(FOLD HERE)

PLACE
POSTAGE
HERE

BOB DIETRICH
M. S. 63-211
TEKTRONIX INC.
P.O. BOX 500
BEAVERTON, OREGON
97077 U.S.A.

(FOLD HERE)

NOTE: Pascal News publishes all the checklists it gets. Implementors should send us their checklists for their products so the thousands of committed Pascalers can judge them for their merit. Otherwise we must rely on rumors.

Please feel free to use additional sheets of paper.

IMPLEMENTATION NOTES ONE PURPOSE COUPON

POLICY: PASCAL USERS GROUP (17-Mar-80)

Purpose: The Pascal User's Group (PUG) promotes the use of the programming language Pascal as well as the ideas behind Pascal through the vehicle of Pascal News. PUG is intentionally designed to be non political, and as such, it is not an "entity" which takes stands on issues or support causes or other efforts however well-intentioned. Informality is our guiding principle; there are no officers or meetings of PUG.

The increasing availability of Pascal makes it a viable alternative for software production and justifies its further use. We all strive to make using Pascal a respectable activity.

Membership: Anyone can join PUG, particularly the Pascal user, teacher, maintainer, implementor, distributor, or just plain fan. Memberships from libraries are also encouraged. See the **ALL-PURPOSE COUPON** for details.

Facts about Pascal, THE PROGRAMMING LANGUAGE:

Pascal is a small, practical, and general-purpose (but not all-purpose) programming language possessing algorithmic and data structures to aid systematic programming. Pascal was intended to be easy to learn and read by humans, and efficient to translate by computers.

Pascal has met these goals and is being used successfully for:

* teaching programming concepts
* developing reliable "production" software
* implementing software efficiently on today's machines
* writing portable software

Pascal implementations exist for more than 105 different computer systems, and this number increases every month. The "Implementation Notes" section of Pascal News describes how to obtain them.

The standard reference and tutorial manual for Pascal is:

> Pascal - User Manual and Report (Second, study edition)
> by Kathleen Jensen and Niklaus Wirth.
> Springer-Verlag Publishers: New York, Heidelberg, Berlin
> 1978 (corrected printing), 167 pages, paperback, $7.90.

Introductory textbooks about Pascal are described in the "Here and There" section of Pascal News.

The programming language, Pascal, was named after the mathematician and religious fanatic Blaise Pascal (1623-1662). Pascal is not an acronym.

Remember, **Pascal User's Group** is each individual member's group. We currently have more than 3357 active members in more than 41 countries. this year Pascal News is averaging more than 120 pages per issue.

Return to:

PASCAL USERS GROUP
P.O. Box 888524
Atlanta, GA 30338

Return postage guaranteed
Address Correction requested

Bulk Rate
U.S. Postage
PAID
Atlanta, Ga.
Permit No. 2854