

ABM \$10.00

PASCAL USERS GROUP

Pascal News

Communications about the Programming Language Pascal by Pascalers

- APL Scanner
- Computer Generated Population Pyramids
- Path Pascal
- Introduction to Modula-2
- Validation Suite Reports
- Announcements

Number

26

JULY 83

POLICY: PASCAL NEWS

(Jan. 83)

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

Purpose: The Pascal User's Group (PUG) promotes the use of the programming language Pascal as well as the ideas behind Pascal through the vehicle of *Pascal News*. PUG is intentionally designed to be non political, and as such, it is not an "entity" which takes stands on issues or support causes or other efforts however well-intentioned. Informality is our guiding principle; there are no officers or meetings of PUG.

The increasing availability of Pascal makes it a viable alternative for software production and justifies its further use. We all strive to make using Pascal a respectable activity.

Membership: Anyone can join PUG, particularly the Pascal user, teacher, maintainer, implementor, distributor, or just plain fan. Memberships from libraries are also encouraged. See the COUPON for details.

- *Pascal News* is produced 4 times during a year; January, April, July October.
- ALL THE NEWS THAT'S FIT, WE PRINT. Please send material (brevity is a virtue) for *Pascal News* single-spaced and camera-ready (use dark ribbon and 15.5 cm lines!)
- Remember: ALL LETTERS TO US WILL BE PRINTED UNLESS THEY CONTAIN A REQUEST TO THE CONTRARY.
- *Pascal News* is divided into flexible sections:

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

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

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

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

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

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

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

Pascal News

Communications about the Programming Language Pascal by Pascalers

JULY 1983

NUMBER 26

2 EDITOR'S NOTES

5 OPEN FORUM

SOFTWARE TOOLS

11 Program APL Scanner

By Vincent Dichristofano, Alan Kaniss, Thomas Robinson and John Santini

ARTICLES

26 "Don't Fail Me Now" By Srully Blotnick

27 Computer Generated Population Pyramids Using Pascal By Gerald R. Pitzl

32 Path Pascal — A Language for Concurrent Algorithms By W. Joseph Berman

37 An Introduction to Modula-2 for Pascal Programmers

By Lee Jacobson and Bebo White

BOOK REPORT

41 Data Structures Using Pascal

ANNOUNCEMENTS

42 SBB Announces Pascal Compiler for IBM PC

42 Sage Opens Boston Division

42 New 16 Bit Sage IV

43 New Modula-2 Manual

44 USUS Fall Meeting

44 Text Editor Interest Group

44 Modula-2 Users Group

45 USUS San Diego Meeting

46 Volitions Modula-2 for IBM PC

47 IMPLEMENTATION REPORT

VALIDATION SUITE REPORT

48 OmegaSoft Pascal Version 2

51 SUBSCRIPTION COUPON

53 VALIDATION SUITE COUPON

55 USUS MEMBERSHIP COUPON

Charles Gaffney Publisher and Editor

The Pascal Newsletter is published by the Pascal Users Group, 2903 Huntington Rd., Cleveland, Ohio 44120. The Pascal Newsletter is a direct benefit of membership in PUG.

Membership dues in PUG are \$25.00 US regular, other forms of membership please inquire. Inquiries regarding membership should be sent to the above address. Newsletter correspondence and advertising should be sent to the editor at the aforementioned address.

Advertising Rates: \$300.00 Full Page. Please give your preference of magazine location: front, center, or back.

Hello,

Well, this is the third issue I am involved with and there have been many changes. I would like to write of Pascal first.

Pascal has enjoyed a jump in attention in the last year. One reason is that there are Pascal compilers available for many machines and, I am tempted to say, they are available for any machine. Most of the major main frames have Pascal either directly or from a third party.

One step down in size, I know of only one machine, the Tandem computer which is without a Pascal implementation. A Tandem representative here in Cleveland informed me they have a language called "TAL" and in many cases will execute a Pascal program with no changes.

A couple more steps down in size are the small Digital Equipment machines and compilers are available from about four sources. IBM has the Display writer and Datamaster. These were released without our language, but in the last year, UCSD Pascal has been made available through IBM. Apple Computer has been a strong and long supporter of Pascal. TRS 80 has UCSD Pascal.

The smallest machine with Pascal is the TI 99/4A. In this size, Commodore has promised Pascal for this summer on the "64" and "128" machines.

The small computer, that is, the home computers and small business computers, have exceeded \$10 million in sales. This is according to Future Computing, a Richardson, Texas research firm.

With a guess, I would say that Pascal is implemented on at least 25% of these machines. If only 1% of these were being used to learn and program Pascal, then 25,000 people are presently involved. This is a lot of people looking for the best books from which to learn.

I am making an appeal to our members to submit comments and reviews of text books so that we all may benefit from your experience. I get calls from authors requesting information on Pascal. To these people, the best I can do is to send complete sets of *Pascal News*! With your comments and criticism, perhaps we could influence future text books.

Herb Rubenstein of Budget Computer in Golden, Colorado has sent a small article from *Popular Computing*. It seems that advanced placement test in computer science will use structure programming and the Pascal language. These tests allow up to one year of college level credits in computer science. The author of this article, Dan Watt, believes that the choice of Pascal in the testing may lead to Pascal as a defacto standard in high schools preparing students for college. Let me quote the last paragraph:

"This situation illustrates the power of the testing establishment to influence the lives of students and teachers. Although the vast majority of high schools now offer Basic as the standard computer language for most programming and computer science classes, this action by the College

Board may lead to the establishment of Pascal as a defacto standard for high school teaching and spawn an entire mini industry of curriculum to meet the new requirements. It may also offer significant school marketing advantages to micro-computer companies that already support Pascal — such as Apple, IBM and Texas Instruments."

I would like to see comments from you regarding this use of Pascal in a rite of passage.

In this issue, you will find a reprint of Dr. Sully Blotnick's column from *Forbes* magazine. I like this column because of the clever way he has made our economy dependent on you learning Pascal.

I enjoy *Forbes* magazine. They emphasize common sense and illustrate proven business practices. *Forbes* also takes a pulse of industries, and small computers is a fast growing industry. In a column called "Technology", edited by Stephen Kindel, on March 28, 1983, he noted that 2% of the households in the U.S.A. own computers of one form or another. There had been predictions of 40% of households by 1990. This has been reduced to 20% in 1990 because there doesn't seem to be software that is useful in households.

Mr. Kindel ends this article with a quote from Seymour Papert, an MIT professor:

"The real purpose of learning how computers work should be to improve human logic and thought processes, to make people more creative, not simply more dependent on machines."

Maybe this would be a good issue to review the tools available in our back issues. This issue contains the APL scanner. I am embarrassed to print this, not because of the program's quality, but because it was submitted four years ago. Well, no time like the present.

In issue #17 (yellow), Arthur Sale submitted "Referencer", a procedural cross reference. This program provides a printout of the heading of each procedure and function with indentation showing nesting. In issue #25, Mr. Yavner has improved on this program with "A Better Referencer". Mr. Yavner claims that *Pascal News* has been his sole source of instruction in Pascal. I believe this is a compliment to Andy Mickel and Rick Shaw for their efforts to maintain this newsletter. We should also thank our contributors, Mr. Sale for instance, for outstanding generosity. These people will appreciate your complements, criticisms and gifts of money. (Ho! Ho!)

Andrew Tandenbaum, in issues 21 and 22/23, provided us with "The EM1 Compiler". This is a good look at all that is necessary for a pseudo 32-bit machine Pascal compiler.

The UCSD Pascal Project started with a 16-bit pseudo machine portable compiler. It was called P4 out of Zurich, Switzerland by Vrs Ammann, Kesav Nori and Christian Jacobi. I mentioned this because it has been published with critical commentary by S. Pemberton and M.C. Daniels in 1982. It is presented as a

case study of compiler design and is very interesting to read.

Pascal Implementation

S. Pemberton and M.C. Daniels
Ellis Horwood Limited Publishers

Distributed by:

John Wiley & Sons
605 Third Avenue
NY, NY 10016
USA

In #21 you will find Jeff Pepper's fine implementation of extended precision arithmetic.

Nicklaus Wirth, Pascal's creator, wrote Pascal S and we have it in #19 (misabeled #17). This is a subset of Pascal and was intended as a teaching aid.

Also in #19 is a Lisp interpreter written in Pascal.

"MAP", a Pascal macro preprocessor for large program development, is published in #17.

Issue #16 contains the Validation Suite version 2.2. This is the compiler checker that Arthur Sale and Brian Wickman have now revised to version 3. This new version is available by using the Validation Suite coupon in the rear of this issue.

"Prose", a text formatter, by John Strait is the major program available in #15. A disclaimer in the instructions manual admits that it doesn't do everything, but I must say, it has a lot of capability.

In #13, two programs were printed that performed the same work. A sort of "Battle of Algorithms". "Pretty Print" and "Format" used any Pascal programs as input and printed it in a consistent style.

For those of you looking for other Pascal periodicals, there are four of which I know. "Pascal Market News", 30 Mowry Street, Mt. Carmel, CT. 06518. This is a nice quarterly for \$9.

Another quarterly for Oregon Software users is the "Pascal Newsletter". Maybe this is too narrow in content, but you will know what Oregon Software is up to. Their address is 2340 SW Canyon Rd., Portland, OR. 97201.

A very slick magazine with good design is "Journal of Pascal and Ada." You can contact them at West Publishing Company, 898 South State Street, Orem, UT. 84057. The cost is \$14 for six issues.

The USUS News and Report is more a system user's journal, but the system is based on Pascal. They also have a software library, seventeen floppy disks full, and all in source code and written in Pascal.

Now to the business of *Pascal News*. *Pascal News*, as the Pascal periodical granddaddy published since January 1974, has had its ups and downs. In 1979 our circulation was 7,000; now it is 3,600. Our biggest problem has been irregular publication. I am committed to four issues this year and I am considering six issues next year. I believe that regularity will supply us with growth and members and more software tools.

As I mentioned in the last issue, PUG (AUS) has stopped and I, in the USA, have taken over their area. Unfortunately, they have not sent me their mailing list and I fear that I have lost touch with our members there. This issue will be sent to those members listed as of 1979 and I hope they will "spread the word" and the subscription coupons!

Open Forum

Our PUG (EUR) has performed very nicely and I thank Helmut Weber and friends for their good work. But they have a problem concerning money. They have not charged enough for subscriptions and were pressed to send our #24. As a result, I will mail all issues directly and I hope you will not be inconvenienced. Please keep in touch with them as they are a strong group.

I have saved the worst for last. In November, 1982, I sent 300 copies of issue #24 to Nick Hughes in care of PUG (UK), Post Office Box 52, Pinnen, Middlesex HA5 3FE, United Kingdom. Using the phone number 866-3816, the air express shipper delivered these issues by mid-November. All well and good. The issues arrived before the cover date with plenty of time to post them to our English members. I called Nick at this number many times, but spoke to him only after many months. It was late April and I asked if I should use the same procedure in shipping #25 to him.

Nick said that the issues arrived properly and that method was efficient but wanted to know what was in #25. He told me that he did not like issue #24 and from the sound of it, did not like issue #25. He had disliked #24 so much, he decided not to send any of them out. Need I say more?

Nick will not supply his mailing list so I am sending this issue and #25 directly to the members of record in the United Kingdom as of 1979. If you feel a need to find out why Nick Hughes did not like issue #24 or would like to see it yourself, please call or write Nick at the above address and ask for your copy. He has 300 and I am sure he can spare one.

As a result of these difficulties, I will receive and service all subscriptions from here in Cleveland, Ohio. From now on, there will be only one person to blame if you have a complaint.

As of this issue, a year's subscription is raised in price to \$25 a year and \$50 for three years. These represent two sets of costs; production and organization. Production costs are typesetting, printing and mailing. Other activities of production are editing, reviewing, quality assurance and formatting. These tasks are performed by "yours truly" and presently I do them for free. (I'm real smart!)

Organization is a cost of servicing you and other members satisfactorily. This includes collecting and reviewing the mail, depositing checks, updating the mailing list, sending back issues to fill new subscriptions and sending sets of previous years back issues. In order to do this correctly, and in a timely fashion, I don't do it. I pay a firm to perform "fulfillment" and it takes one or two days per week. This cost is small compared to the bad feelings generated if not done correctly and quickly.

These are costs of which you are totally responsible. This newsletter has been a beneficiary of volunteerism. There are no volunteers now (save me). In many magazines, advertisements will pay for all production and organizational costs plus provide profits, sometimes large profits.

The cost of a full page ad in *Byte* or *PC* or *PC World* is over \$2,000 and these are publications with 500 pages!

Now we may be able to keep our costs down and publish more often if we accept advertising. Three hundred dollars per page is not expensive. I will pursue

advertisers and I am asking for your help. If you are writing a book, have your publisher advertise with *Pascal News*. If you are making software packages, influence your boss in the virtues of an ad in *Pascal News*. If you manufacture or sell computers, sell your product from the pages of *Pascal News*. This is the oldest Pascal publication and, I proudly say, the most influential.

This newsletter help spread Pascal and our members were most influential in the standard efforts.

I believe *Pascal News*' new mission is to enable Pascal to be taught in the easiest way. This is in many forms. For instance, reviews of books and texts, discussion of what features to teach first as a foundation, how to teach advanced courses, discussions of extensions or standard program tools to include in every well written program as it is appropriate.

By the way, Andy Mickel tells me that the "Pascal User's Manual and Report" by Jensen and Wirth has sold 150,000 copies in 1982. This is interesting considering that in the previous seven years, it sold 175,000 copies. A very sharp jump in interest.

A new text book has been sent to me, "Pascal" by Dale/Orshalik, 1983 DC Heath. A nice title, short and to the point. The preface states a philosophy that I would like you to comment on.

"In the past there have been two distinct approaches used in introductory computer science texts. One approach focused on problem solving and algorithm design in the abstract, leaving the learning of a particular language to a supplemental manual or a subsequent course. The second approach focused on the syntax of a particular programming language, and assumed that the problem-solving skills would be learned later through practice.

We believe that neither approach is adequate. Problem solving is a skill that can and should

be taught — but not in the abstract. Students must be exposed to the precision and detail required in actually implementing their algorithms in a real programming language. Because of its structured nature, Pascal provides an effective vehicle for combining these two approaches. This book teaches problem-solving heuristics, algorithm development using top-down design, and good programming style concurrently with the syntax and semantics of the Pascal language."

One of the letters mentions high resolution graphics. I know of two texts that use Pascal as the illustrative language of their algorithms. They are "Principles of Interactive Computer Graphics" by Williams Newman and Robert Sproull, 1979 McGraw-Hill and "Fundamentals of Interactive Computer Graphics" by James Foley and Andries Van Dam, 1982 Addison-Wesley.

Two notes from members:

Steven Hull of Campbell, California, received a notice from me that #22/23 had been returned to us because the postal service will not forward bulk mail. His reply:

"I guess this will teach me to move from Lakewood (a suburb of Cleveland, Ohio). Didn't know bulk mail wasn't forwardable. The Postal Diservice has been re-routing every piece of junk mail for a full year . . . I might have to file suit to stop it all!

And from Eric Eldred of New Hampshire who rewarded *Pascal News* with a three year subscription and dutifully filled the coupon with name and address and arrived at a request for "Date". Eric filled in "No! Married!". Thanks Eric, I needed that!

Charlie

To Charlie Gaffney,

I'm glad you have taken on *Pascal News*. I hope it works.

Perhaps, I should say what I would like to see published in *Pascal News*. The most valuable things are 1) Tools, and 2) Info on the various implementations. In my job we are using many computers. It is very helpful to know which compilers work well, meet standards, and produce efficient code. Apple Pascal is nearly bug free, and works as specified (with UCSD quirks). IBM Pascal VS is good — extensions are large presenting conversion problems if they are used. It has a good interface to FORTRAN. VAX Pascal is plain vanilla, appears to work well but we have not tested it in difficult situations. HP Pascal 1000 works fine but does not have a stack architecture and seems to compile slowly. Recent tests on HP Pascal 1.0 for the HP 200 computers seem to indicate it derives from UCSD although it is a native code 68000 compiler. It seems to work very well. We are interested in Pascal for the Data General Eclipse.

Good luck,
Dennis Ehn
215 Cypress Street
Newton Centre, MA 02159

Gentlemen:

Would you be so kind as to send information on the Pascal User's Group (PUG) and its official publication *Pascal News*. Recently we have acquired a microcomputer Pascal compiler and are very much interested in keeping up with current developments in Pascal.

Our system is based upon a SouthWest Technical Products Corporation S/09 computer, running the UniFLEX Operating System (similar to UNIX). If specific information is available for this unit, please let us know.

Additionally, the college has several (approximately 18) Apple computers which are capable of running the UCSD Pascal System. Once again, any special information here would be very helpful.

We look forward to hearing from you and hope that we can make a positive contribution to the Pascal User's Group.

Yours Truly,
Lawrence F. Strickland
Dept. of Engineering Technology
St. Petersburg Jr. College
P.O. Box 13489
St. Petersburg, FL 33733

Dear Sir,

I just received issue number 25 of *Pascal News* and was surprised to find an implementation note for our

Open Forum

Pascal compiler. What makes it surprising is that to the best of my knowledge I have never sent in an entry, and the information provided is about a year and a half out of date.

In case you would like to provide your readers with valid information, I have enclosed an implementation note for the currently available compiler. I have also enclosed a copy of the ISO validation suite report from our language manual.

Work is currently being done on moving this compiler to the 68000 family of processors and should be available by the end of 1983.

On another note, I have received issues number 21, 22/23, and 25, but not issue 24. I am also enclosing a check for a 3 year membership — please see if you can determine what happened to number 24.

Sincerely,
Robert Reimiller
Owner, OmegaSoft
5787 Brandywine Ct.
Camarillo, CA 93010

December 1, 1982

I hope the letter referring to the possible end of the P.U.G. is wrong! I can be of some help if needed.

Allen Duberstein
Pine Instrument Co.
3345 Industrial Blvd.
Bethel Park, PA 15102

January 10, 1983

Dear Mr. Gaffney:

Enclosed is a check covering both the remailing cost of *Pascal News* #24 (\$5) plus my membership renewal for two years (\$18).

My apologies for getting out of synchronization with the Pascal Users Group. As the post office informed you, I recently moved to the address noted. Frankly, I hadn't received a *Pascal News* in so long that I simply forgot about it. It appears that I won't miss any issues — the enclosed All-Purpose Coupon is from issue #23.

Interestingly, after a long period (3 years) of not using Pascal, it looks like I will be using it once again. We have a couple of Convergent Technologies workstations in my office. These are very nice 8086-based machines; Burroughs sells them as the B-20s, and NCR sells them as WorkSavers. We will probably be getting a Pascal compiler, and I am looking forward to getting back into Pascaling in the near future.

Sincerely,
Read T. Fleming
144 Irving Avenue #B-3
Providence, RI 02906

November 30, 1982

I was surprised and pleased to receive issue number 24 of *Pascal News*. Thanks for taking it over. I do have one question, however, which you might be able to help me with. What year is it? My address label includes [82] on it but the previous issue I received was dated September, 1981. I notice that this issue is dated January, 1983. Should I send in another year's subscription money now? What happened to 1982? I never have managed to figure our *Pascal News*' subscription scheme. Maybe a note in the issues towards the end of a year saying "if your address label says [82] it's time to send in a renewal" would help.

Thanks for your help.

Richard Furuta
Computer Science, FR-35
University of Washington
Seattle, WA 98195

8 February 1983

Dear Sir,

I received your notification of renewal in the mail yesterday. I am slightly concerned that you may not have received the check which I mailed to you in December. I hope that it has only been a slight mix-up, and in fact, my subscription has been renewed for 3 years, as I requested.

I am currently using the Pascal implemented by Microsoft for the IBM Personal Computer. It has some non-standard features which were provided in order to allow programmers to access the full capabilities of the machine. This implementation is quite flexible, and was designed to allow users to produce systems programs, as well as application programs.

The greatest shortcoming to this product, however, is its lack of usable documentation. Even someone like myself, who has been programming in Pascal for 8 years, has difficulty in trying to locate the appropriate material in the 'reference manual'. Once this is overcome, the user is able to use this version for the production of some very powerful software.

I continue to look forward to the delivery of your fine newsletter. I enjoy the articles, and realize how difficult a task you have. Keep up the good work.

Regards,

Robert A. Gibson
1609 Lake Park Dr.
Raleigh, NC 27612

November 30, 1982

Pascal is being used for process control of laser trimming systems. We use Oregon Software Pascal.

Barbara Huseby, Training Dept.
Electro Scientific Industries
13900 N.W. Science Park Drive
Portland, OR 97229

March 3, 1983

Dear Mr. Gaffney:

I'm writing to let you know why I am not renewing my subscription to *Pascal News*. The main reason is that the price is now too high for the utility of the product (at least to me). I appreciate your efforts to keep PUG and *Pascal News* going, but I'm afraid they may have outlived their usefulness. Pascal is not really in need of promotion as it was when PUG was formed. The *Journal of Pascal & Ada* may be an appropriate successor.

As a long-time subscriber and occasional contributor, I wish you luck in your efforts.

Richard Leklanc
Assistant Professor
Georgia Institute of Technology
Atlanta, GA 30332

January 7, 1983

Hang in there, Charlie!

Andy Mickel
106 SE Arthur Avenue
Minneapolis, MN 55414

December 9, 1982

Dear Sirs:

Could you provide us with information on membership in your organization, both personal and institutional, as well as the subscription cost of your journal.

We are also interested in a rigorous comparison of the various PASCAL versions implemented by mini and microcomputer vendors. Do you know of any such comparative research? We are making plans to offer Advanced Placement Computer Science in the fall term of 1983, and wish to select an effective computer.

Very truly yours,

Charles McCambridge
Director
Instructional Materials Services
Niskayuna High School
1626 Balltown Rd.
Schenectady, NY 12309

December 25, 1982

Merry Xmas! Good luck, Charlie! Is your "acquisition" of PUG a sign that PUG and USUS will someday merge? I'm not sure I'd like that, but let's see.

Jim Merritt
P.O. Box 1087
Morro Bay, CA 93442

December 24, 1982

Please send me information on joining the Pascal
Open Forum

User's Group, I am a software project engineer at General Electric in Syracuse. I am currently in the process of selecting a high level language for internal programming of a 1024 x 1280 resolution raster display. Pascal is the leading candidate, therefore, I am very interested in the latest information regarding the language which I feel a user's group could provide.

My interest does transcend my work however as I do own a Commodore SuperPET which includes the University of Waterloo software package consisting of Pascal, APL, Fortran, Basic and a 6809 Assembler.

Sincerely,

Douglas W. MacDonald
4303 Luna Course
Liverpool, NY 13088

2/5/83

To Whom It May Concern:

I just received your notice to inform me that my membership is about to expire and that I should renew now.

I would like to tell you that I would consider renewing if I could be assured of getting my money's worth — this time!

When I first joined in 1981, I didn't hear from *Pascal News* for almost a year. Then a few months ago, I received a second issue, but that's been it.

Now I am a convicted Pascaler. I understand the difficulties of operating a non centralized club, but \$20 should buy *some kind* of organization for things I feel.

Can you assure me of a better value this time around?

Cordially,

David Abate
Micro People
116 S. Bowdion St.
Lawrence, MA 01843

P.S. Question: Do you intend anything on UCSD-Pascal? This is my greatest interest.

7 January 1983

Hi,

This is a note in a bottle to: 1) find out if you're still out there, and 2) what's happening with Pascal. It doesn't seem to be taking the bite (or is that byte) out of Basic I thought it would.

We will start covering Pascal as soon as we have finished Basic programming — about five weeks from now. The extension program from Hocking Technical College in Nelsonville has provided seven Apple II and Apple III computers and two printers. By the end of the year, they will have installed a winchester disc and either a modem or a microwave link to their main campus computer. We'll need it by then to cover the Cobol and Fortran IV programs we'll be writing.

Most of my practical computer experience is in assembler language. I used it at Cincinnati Milacron's

Process Controls Division (Mater's of the controls for the T³ Industrial Robot).

I am interested in any literature you have to send me. In particular, I would like the titles of the books you consider best for teaching Pascal — either on the Apple II or on computers in general. Apple, Inc., sent me the Pascal Reference Manual (just a bit or a nibble over my head). I've also read copies of the DOS 3.2 Reference Manual and their Basic Programming Manual. I covered all these before classes started and wound up tutoring two other student/inmates.

Sincerely,

Brian Appleman 166-767
15802 St. Rt. 104
P.O. Box 5500
Chillicothe, OH 45601

P.S. If you need more on my background, just ask.

83-02-24

Dear Charlie:

I am a member of PUG (AUS) which has just folded, and I would like to re-enroll through PUG (US).

I don't share Arthur Sales view that PUG and PN have no purpose now that there is an ISO standard. The world still needs cheap, good software and PN (in a modest way) supplies some of it. Also, some organization is needed to defend and develop good programming language and style.

PUG (AUS) says I have a credit of 12 (old) issues and that the funds have been sent to you. Please will you accept my re-enrollment and advise me how many (new) issues I am now entitled to?

Finally, I, and I'm sure, many others appreciate your offer to keep PUG/PN going.

Thanks again.

Yours sincerely,

Peter Edwards
40 Davison St.
Mitcham, Vic.
Australia 3132

December 3, 1982

Best wishes in this venture, Charlie. I agree that *Pascal News* and P.U.G. are worth saving.

John W. Baxter
750 State Street, Apt. #224
San Diego, California 92101

February, 1983

You people have ripped me off for the last time!

By your own back order form (attached) you show that my renewal in 1981 paid for 3 issues mailed in 1982. But then, WHAT OF MY RENEWAL PAID IN 1982? ONLY ONE ISSUE #24 COUNTS??? AND THAT HAD TWO PREVIOUSLY PUBLISHED PRO-

GRAMS!! (That is, programs I had ALREADY received.) If you ran a decent organization, you'd make my 1982 renewal count for 1983 also.

David S. Bakin
Softech Inc.
360 Totten Pond Road
Waltham, MA 02154

December 24, 1982

We're indebted to you, Charlie!

Wayne N. Overman
3522 Rockdale Ct.
Baltimore, MD 21207

February 17, 1983

Dear Mr. Gaffney,

I am one of those folks who does not have a currently correct address with *Pascal News*.

Enclosed is a check for \$5 for a copy of issue 19 which was returned to you.

Thank you on behalf of all the members of the user's group for the effort you are putting out. It is very much appreciated.

Tom Bishop
P.O. Box A
Kenmore, WA 98028

March 14, 1983

Dear Sir or Ms.:

We plan to offer Pascal at our school. I would appreciate receiving information on your group and, if possible, a sample copy of *Pascal News*.

Any suggestions or information you could send would be appreciated. We are particularly concerned that the new Apple 2-E does not support Pascal with one disk drive. We had hoped tht UCSD Pascal with one drive would work on the Apple 2-E.

Thanks for your help.

Sincerely,
Harold Baker
Director, Computer Science
Litchfield High School
Litchfield, CT 06759

February 11, 1983

Hi!

Here's my renewal. I really enjoy *Pascal News* and have been upset about what has happened with it the past 18 months or so. It has been of substantive value to me, particularly in the area of the style of Pascal coding among the community that have submitted articles.

I would like to see more articles on Modula 2,

Wirth's follow on to Pascal and Ada in parallel. To me, this would seem a way of keeping PUG alive as well as providing a growth path to these languages for Pascal programmers.

I use Pascal/VS extensively at work and I have found its extensions the best of any other Pascal compiler for S/370 compatible machines. Almost all of its extensions are within the "spirit" of Pascal and uses a very good extension to STRING data. Of particular convenience is its READSTR and WRITESTR functions (they are procedures actually -unfortunately). I force the concept of function upon them by embedding their invocation within a function when required.

I never received issues 20 and 21 of *Pascal News* during the confusion, although I did mention this at times. I would certainly purchase them separately, but I am not prepared to purchase two sets to get them. Please advise.

Thanks for your work,

Bob Dinah
630 Alvarado St. #207
San Francisco, CA 94114

November 12, 1982

Dear Pascal User's Group:

The only source of information that I have on the Pascal User's Group came from "The BYTE Book of Pascal", according to an article written by Kenneth Bowles. An editor's note of July 1, 1979 listed the annual newsletter as \$6.00 per year. I am enclosing \$12.00 in case things have increased since that date. If this amount is insufficient, please make it up on back issues.

I am currently using an Apple III with Apple computer's version of UCSD Pascal. There does not seem to be more than a dozen books written on Pascal, and just a few on UCSD.

I am an ex-electrical engineer, turned to building construction. Previously, I worked for Westinghouse Research Center in Pittsburgh, and used the Burroughs B6500 main frame computer with ALGOL language. The B6500 used a number of formats and types that I miss; the Fixed Format was especially useful since it allowed the user to specify the number of total digits and the number of decimal digits combined. I would like to use this format in UCSD Pascal.

Thanks for taking the time to help me.

Very truly yours,
Larry J. Moorhead
5207 - 32nd Street East
Bradenton, Florida 33508

18 March 1983

Dear Sirs,

For the first time we have received a copy of *Pascal News*, and it has been read with great interest.

We would like to join your User Group but cannot find either a price or contact address for our region.

Please send us this information as soon as possible,

so that we can become members and start receiving your journal on a regular basis.

We have taken note of your abhorrence of paper-work (and endorse the sentiment) and will send the necessary prepayment once we receive the information.

Yours sincerely,

Bette Kun
Librarian
Control Data
P.O. Box 78105
Sandton, South Africa 2146

20 April 1983

Dear Mr. Shaw:

Enclosed is a check for \$10.00 for a one-year subscription to the PASCAL Users' Group Newsletter. We have just recently acquired PASCAL-2 here at Villanova and our students are using it on LSI-11 systems running RT-11 V4.0 for applications involving real-time control, data acquisition, and computer communications.

Sincerely yours,

Richard J. Perry, Ph.D.
Villanova University
Dept. of Electrical Engineering
Villanova, PA 19085

15th February, 1983

Dear Mr Gaffney,

As a long PUG user the demise of PUG-AUS is a blow. Anyhow, as you can see from the attached letter I would *love to continue* and thus need your help.

Could you please detail the fees for 1983 for us "down under" for surface mail and air mail and as you can see I'm afraid I've not got issue number 21. Can you help?

For interest I use:

UCSD Pascal/p-System	ERA-50 Computer (8-bit, 8085 base)
Pascal MT+ under CP/M and MP/M	ERA-50 Computer (8-bit, 8085 base) ERA-50 Computer (8-bit, 8085 base)
Pascal MT+ 86 under CP/M-86 and MP/M-86	ERA-80 Computer (16 bit, 8086/8087 based) ERA-80 Computer (16 bit, 8086/8087 based)

Regards,
Dr. William J. Caelli, F.A.C.S.
President
ERACOM Group of Companies
P.O. Box 5488, G.C.M.C.
Qld. 4217, Australia

1-8-82

Dear Sir/Madam,

This is the first letter I write to contact you. Let
Open Forum

me introduce myself first. I am a student pursuing a computer course in the Hong Kong Polytechnic — a licensed user of your OMSI-PASCAL-2 V1.2. I don't know what your definition of user may be. May it be my Polytechnic or any student or programmer who use your OMSI PASCAL-1 under the Polytechnic, I venture to call myself a user in this letter, and would like to join the Pascal Users' Group and receive the newsletter.

In the past few months, I have been doing extensive programming using PASCAL, and find it very handy, especially in writing structured programs. However, until recently when I develop some system programs, I find problems. I discover that there is no source listing or documentation on the OMSI PASCAL-1 run time system (possibly in file FPP.RTS) and its relationship with RSTS/E, and I cannot interface with the low level I/O trap handlers without knowing their details. I find some problems on the RESET ODT mode, but I cannot deal with it in assembly level.

All in all, my problem is highly personal and does not in any way bear relation with the Hong Kong Polytechnic. However, as a student on computing, I don't want to leave problem unsolved. So, please send me any informational help, if possible.

Included please find a bank draft of \$6 for subscription.

May I state once more my request. I need information on OMSI PASCAL-1 run time system especially the EMT trap handling.

Thank you very much in advance.

Yours faithfully,

Mr Kam Man-Kai
Flat 8
3/F Ting Yin House
Siu On Court
Tuen Mun - N.T.
Hong Kong

6th November, 1982.

Dear Mr. Mickel,

I am a student of computing studies in the H.K. Polytechnic. Recently, I got a chance to buy a Chinese version of 'A Practical Introduction to Pascal' by Wilson & Addyman from which I was informed that there is a PUG in States.

Briefly understanding the objectives of the PUG, I find myself in great interest in joining the group. Would you be so kind as to provide me with further information as far as the PUG is concerned. I am eagerly looking forward to your reply.

Yours sincerely,

Alan Kwong
12, Boundary St.
Po Hing Bldg.
8/F, Block 'C'
Kln., H.K.

December 23, 1982

We have been using Oregon Software's RT-11 Pascal implementations for over three years with excellent results and complete satisfaction; Pascal is used for scientific "number crunching", program development, algorithm testing, etc.

Bob Schor
The Rockefeller University
1230 York Avenue
New York, NY 10021

December 30, 1982

A worthwhile journal.

George Williams
Union College
Schenectady, NY 12308

March 22, 1983

Dear Mr. Gaffney:

I have previously received *Pascal News* through University of Tasmania. Is it still published? If so, do I have any credit on my subscription dues? I would also be interested in information about USUS.

Yours sincerely,
M.J. Palmer
CSIRO
Private Bag
P.O. Wembley, W.A. 6014

February 3, 1983

Good job, Charlie! and good luck to the renewed *Pascal News*!

Norman W. Molhant
320 Principale
Tres-Saint-Redempteur, P.Q.
Canada J0P 1P0

May 1, 1983

A professor in Ithaca, NY told me there exists a public domain UCSD Pascal available for micro's.

I have a 60K Z-80 which uses memory map video, and a 63K 8085/8088 (both machines S-100 bus) which uses a TVI 950. I also have a H-29 terminal (like Z-19 but with a detached keyboard).

Is there really any way of getting this UCSD Pascal running on one of my systems? (I have UCSD on the Sage also Modula-2. Good stuff.)

Thanks,
J. E. Pournelle, Ph.D.
12051 Laurel Terrace
Studio City, CA 91604

Program APLscanner

By Vincent Dichristofano, Alan Kaniss, Thomas Robinson, and John Santini
NADC, Philadelphia, PA

```

1 program APLscanner(input ( + TERMINAL ), output , APLfile );
2
3 {
4 * Purpose:
5   This program is an implementation of APL in Pascal.
6
7 * Authors:
8   Vincent Dichristofano
9   Alan Kaniss
10  Thomas Robinson
11  John Santini
12    authors' affiliation - NADC
13
14                               Phil. PA. USA
15
16   project leader: Dr. Joseph Mezzaroba
17
18   This program was written as part of an independent study
19   course at Villanova University.
20
21 * Submitted and accepted for Pascal News. DEC 1976.
22
23 }
24 Label
25 100;
26
27 const
28   prefix1 = 60;
29   prefix2 = 62 { prefix for CDC ASCII 12-bit codes };
30   MaxVarNameLength = 10;
31   MaxInputLine = 132;
32   InputArraySize = 134;
33   NumberOfMessages = 100;
34   MessageLength = 80;
35
36 type
37   PackedString = packed array [1 .. MaxVarNameLength] of 0 .. 8191;
38   TokenNoun =
39     (FormRes, FormArg, GlobVar, MonadOper, ReductOper, DyadOper,
40      SpecOper, constant, StatEnd);
41   values = record
42     RealVal: real;
43     NextValue: ^values
44   end;
45   VarTab = record
46     VarName: PackedString [ v1 ];
47     FuncTabPtr: ^FuncTab [ v2 - ftab ];
48     ValTabPtr: ^ValTab [ v3 - vtab ];
49     DeferredValTabPtr: ^FParamTab;
50     NextVarTabPtr: ^VarTab
51   end;
52   ValTab = record
53     IntermedResult: Boolean;
54     dimensions: integer;
55     FirstDimen: ^DimenInfo;
56     ForwardOrder: Boolean;
57     FirstValue: ^values;
58     NextValTabLink: ^ValTab
59   end;
60   TokenTable = record
61     NextToken: ^TokenTable;
62     case noun: TokenNoun of [ p ]
63       FormRes, FormArg, GlobVar: { vtab }
64         (VarTabPtr: ^VarTab);
65       MonadOper: (NonIndex: integer);
66       ReductOper: (RedInIdx: integer);
67       DyadOper: (DOPInIdx: integer);
68       SpecOper: (CharInIdx: integer);
69       constant: (ValTabPtr: ^ValTab);
70       StatEnd: (EndAdj: integer)
71     end;
72   vfunc = record
73     NextStant: ^TokenTable;
74     NextVFuncPtr: ^vfunc;
75     StatLabel: PackedString
76   end;
77   OperatorType = (niladic, monadic, dyadic);
78   FuncTab = record
79     FuncName: PackedString [ f1 ];
80     arity: OperatorType [ f2 ];
81     result: Boolean [ f3 true = explicit ];
82     ResultName: PackedString [ f4 ];
83     LeftArg: PackedString [ f5 ];
84     RightArg: PackedString [ f6 ];
85     FirstStatement: ^vfunc;
86     NextFuncTabPtr: ^FuncTab;
87     NumOfStatements: integer
88   end;
89   FParamTab = record
90     PtrVal: ^ValTab [ sd1 and sd2 ];
91     LastParm: ^FParamTab [ link to last ]
92     [ sd1 or sd2 ]
93   end;
94   DimenInfo = record
95     NextDimen: ^DimenInfo;
96     dimenlength: integer
97   end;
98   OpRecord = record
99     OpIndex: integer;
100    OpSymbol: integer
101  end;
102  OperandTab = record
103    OperPtr: ^ValTab [ sval ];
104    LastOper: ^OperandTab [ link to last sval ]
105  end;
106  SubrTab = record [ sf ]
107    CalledSubr: ^FuncTab [ s1 ];
108    TokenCallingSubr: ^TokenTable [ s2 ];
109    StateCallingSubr: ^vfunc [ s3 ];
110    LastSubrPtr: ^SubrTab [ link to last sf ]
111  end;
112  OpTable = array [1 .. 16] of OpRecord;
113  VarTabPtrType = ^VarTab;
114  TypeValTabPtr = ^ValTab;
115  TokenPtr = ^TokenTable;
116  PtrFuncTab = ^FuncTab;
117  TypeValuesPtr = ^values;
118  APLcharSet =
119    (ASymbol, BSymbol, CSymbol, DSymbol, ESymbol, FSymbol, GSymbol,
120     HSymbol, ISymbol, JSymbol, KSymbol, LSymbol, MSymbol, NSymbol,
121     OSymbol, PSymbol, QSymbols, RSymbols, SSymbols, TSymbols, USymbols,
122     VSymbols, WSymbols, XSymbols, YSymbols, ZSymbols, OneSymbol, TwoSymbol,
123     ThreeSymbol, FourSymbol, FiveSymbol, SixSymbol, SevenSymbol,
124     EightSymbol, NineSymbol, ZeroSymbol, colon, RightArrow, LeftArrow,
125     SmallCircle, period, LeftParen, RightParen, LeftBracket,
126     RightBracket, semicolon, quadrangle, space, plus, minus, times,
127     divide, asterisk, iota, rho, comma, tilde, equals, NotEqual,
128     LessThan, LessOrEqual, GreaterOrEqual, GreaterThan, AndSymbol,
129     OrSymbol, ceiling, floor, LargeCircle, ForwardSlash, DoubleQuote,
130     negative, QuestionMark, omega, epsilon, UpArrow, DownArrow, alpha,
131     UnderScore, del, delta, SingleQuote, EastCap, WestCap, SouthCap,
132     NorthCap, ibeam, TBeam, VerticalStroke, BackwardSlash);
133
134   text = file of char;
135
136 var
137   XColonSym, XRightArrow, XLeftArrow, XLittleCircle, XPeriod, XLeftPar,
138   XRightPar, XLeftBracket, XRightBracket, XSemicolon, XQuadSym:
139   integer;
140   character: array [APLcharSet] of integer;
141   APLstatement: array [1 .. InputArraySize] of integer;
142   digits: array [OneSymbol .. ZeroSymbol] of integer;
143   ErrorMessage: packed array [1 .. NumberOfMessages, 1 .. MessageLength] of
144   char;
145   APLfile: text;
146   MOPTab, DOPTab, RedTab, CharTab, SpecTab: OpTable;
147   SaveLabel: PackedString;
148   name: PackedString;
149   NewTokenPtr, OldTokenPtr, HoldTokenPtr, SaveTokenPtr: ^TokenTable;
150   TestFuncPtr, NewFuncTabPtr, OldFuncTabPtr: ^FuncTab;
151   NewVarTabPtr, OldVarTabPtr: ^VarTab;
152   LeftValPtr, RightValPtr, ValPtr: ^values;
153   NewValues, NewValPtr: ^values;
154   NewDim: ^DimenInfo;
155   DimPtr, NewPtr, LeftDimPtr, RightDimPtr: ^DimenInfo;
156   VarPointer: ^VarTab;
157   OldVFuncPtr, NewVFuncPtr: ^vfunc;
158   NewValTabLink, OldValTabLink: ^ValTab;
159   position: integer;
160   LineLength: integer;
161   code, ColCnt: integer;
162   FuncStatements: integer;
163   TokenError, FirstFunction: Boolean;
164   LineTooLong, HasLabel: Boolean;
165   switch, FunctionMode, TokenSwitch, ItsAnIdentifier: Boolean;
166   OperTabPtr: ^OperandTab [ sv ];

```

```

167 PtrLastOper := "OperandTab;
168 SubrTabPtr := "SubrTab;
169 RParamPtr := "FParamTab | p1 |;
170 LParamPtr := "FParamTab | p2 |;
171 VFuncPtr := "vfunc | nl |;
172 hold := "TokenTable | holds last symbol |;
173
174
175 procedure InitParser;
176
177 begin
178   OperTabPtr := nil; SubrTabPtr := nil; LParamPtr := nil;
179   RParamPtr := nil; VFuncPtr := nil; hold := nil; XColonSym := 1;
180   XRightArrow := 2; XLeftArrow := 3; XLittleCircle := 4;
181   XPeriod := 5; XLeftPar := 6; XRightPar := 7;
182   XLeftBracket := 8; XRightBracket := 9; XSemicolon := 10;
183   XQuadSym := 11; new(OperTabPtr); OperTabPtr.LastOper := nil;
184   PtrLastOper := OperTabPtr;
185 end { initparser };
186
187
188 procedure InitializeCharacterSet
189   { read installation character set from file };
190
191 var
192   TestForPrefix: integer;
193   FileCharacter: char;
194   SymbolIndex: APLcharSet;
195
196 begin
197   reset(APLfile);
198   for SymbolIndex := a symbol to BackwardSlash do
199     begin
200       read(APLfile, FileCharacter);
201
202       { The following code would be removed for non-CDC installations }
203
204       TestForPrefix := ord(FileCharacter);
205       if (TestForPrefix = prefix1) or (TestForPrefix = prefix2)
206       then
207         begin
208           read(APLfile, FileCharacter);
209           character[SymbolIndex] := 100 * TestForPrefix + ord(
210             FileCharacter);
211         end
212       else
213         character[SymbolIndex] := ord(FileCharacter)
214       end
215     end
216   end { initializecharacter set };
217
218
219 procedure ReadInErrorMsgs;
220
221 var
222   MsgRow, MsgCol: integer;
223
224 begin
225   readln(APLfile);
226   for MsgRow := 1 to NumberOfMessages do
227     for MsgCol := 1 to MessageLength do
228       ErrorMsgs[MsgRow, MsgCol] := ' ' { blank out error messages };
229     for MsgRow := 1 to NumberOfMessages do
230       begin { read in error messages from file }
231         MsgCol := 0;
232         while not eoln(APLfile) do
233           begin
234             MsgCol := MsgCol + 1;
235             read(APLfile, ErrorMsgs[MsgRow, MsgCol]);
236           end;
237         readln(APLfile);
238       end
239     end { readinerrormsgs };
240
241
242
243 procedure FillUpTables;
244
245 begin
246   {
247     monadic operators
248   }
249
250   MOpTab[1].OpSymbol := character[plus]; MOpTab[1].OpIndex := 2;
251   MOpTab[2].OpSymbol := character[minus]; MOpTab[2].OpIndex := 3;
252   MOpTab[3].OpSymbol := character[times]; MOpTab[3].OpIndex := 4;
253   MOpTab[4].OpSymbol := character[divide]; MOpTab[4].OpIndex := 5;
254   MOpTab[5].OpSymbol := character[asterisk]; MOpTab[5].OpIndex := 6;
255   MOpTab[6].OpSymbol := character[iota]; MOpTab[6].OpIndex := 21;
256   MOpTab[7].OpSymbol := character[rho]; MOpTab[7].OpIndex := 22;
257   MOpTab[8].OpSymbol := character[comma]; MOpTab[8].OpIndex := 23;
258   MOpTab[9].OpSymbol := character[tilde]; MOpTab[9].OpIndex := 1;
259
260   {
261     dyadic operators
262   }
263
264   DOpTab[1].OpSymbol := character[plus]; DOpTab[1].OpIndex := 52;
265   DOpTab[2].OpSymbol := character[minus]; DOpTab[2].OpIndex := 53;
266   DOpTab[3].OpSymbol := character[times]; DOpTab[3].OpIndex := 54;
267   DOpTab[4].OpSymbol := character[divide]; DOpTab[4].OpIndex := 55;
268   DOpTab[5].OpSymbol := character[asterisk];
269   DOpTab[5].OpIndex := 56; DOpTab[6].OpSymbol := character[iota];
270   DOpTab[6].OpIndex := 87; DOpTab[7].OpSymbol := character[rho];
271   DOpTab[7].OpIndex := 88; DOpTab[8].OpSymbol := character[comma];
272   DOpTab[8].OpIndex := 89; DOpTab[9].OpSymbol := character[equals];
273   DOpTab[9].OpIndex := 71;
274   DOpTab[10].OpSymbol := character[notEqual];
275   DOpTab[10].OpIndex := 72;
276   DOpTab[11].OpSymbol := character[lessThan];
277   DOpTab[11].OpIndex := 73;
278   DOpTab[12].OpSymbol := character[lessOrEqual];
279   DOpTab[12].OpIndex := 74;
280   DOpTab[13].OpSymbol := character[greaterOrEqual];
281   DOpTab[13].OpIndex := 75;
282   DOpTab[14].OpSymbol := character[greaterThan];
283   DOpTab[14].OpIndex := 76;
284   DOpTab[15].OpSymbol := character[andSymbol];
285   DOpTab[15].OpIndex := 77;
286   DOpTab[16].OpSymbol := character[orSymbol];
287   DOpTab[16].OpIndex := 78;
288
289   {
290     special character
291   }
292
293   CharTab[1].OpSymbol := character[colon];
294   CharTab[2].OpSymbol := character[rightArrow];
295   CharTab[3].OpSymbol := character[leftArrow];
296   CharTab[4].OpSymbol := character[smallCircle];
297   CharTab[5].OpSymbol := character[period];
298   CharTab[6].OpSymbol := character[leftParen];
299   CharTab[7].OpSymbol := character[rightParen];
300   CharTab[8].OpSymbol := character[leftBracket];
301   CharTab[9].OpSymbol := character[rightBracket];
302   CharTab[10].OpSymbol := character[semicolon];
303   CharTab[11].OpSymbol := character[quadrangle];
304   CharTab[12].OpSymbol := character[space];
305   Spectab[1].OpSymbol := character[colon];
306   Spectab[2].OpSymbol := character[rightArrow];
307   Spectab[3].OpSymbol := character[leftArrow];
308   Spectab[4].OpSymbol := character[leftParen];
309   Spectab[5].OpSymbol := character[semicolon];
310   Spectab[6].OpSymbol := character[leftBracket];
311
312   {
313     reduction operator
314   }
315
316   RedTab[1].OpSymbol := character[plus]; RedTab[1].OpIndex := 2;
317   RedTab[2].OpSymbol := character[minus]; RedTab[2].OpIndex := 3;
318   RedTab[3].OpSymbol := character[times]; RedTab[3].OpIndex := 4;
319   RedTab[4].OpSymbol := character[divide]; RedTab[4].OpIndex := 5;
320   RedTab[5].OpSymbol := character[asterisk]; RedTab[5].OpIndex := 6;
321   RedTab[6].OpSymbol := character[equals]; RedTab[6].OpIndex := 21;
322   RedTab[7].OpSymbol := character[notEqual];
323   RedTab[7].OpIndex := 22;
324   RedTab[8].OpSymbol := character[lessThan];
325   RedTab[8].OpIndex := 23;
326   RedTab[9].OpSymbol := character[lessOrEqual];
327   RedTab[9].OpIndex := 24;
328   RedTab[10].OpSymbol := character[greaterOrEqual];
329   RedTab[10].OpIndex := 25;
330   RedTab[11].OpSymbol := character[greaterThan];
331   RedTab[11].OpIndex := 26;
332   RedTab[12].OpSymbol := character[andSymbol];
333   RedTab[12].OpIndex := 27;
334   RedTab[13].OpSymbol := character[orSymbol];
335   RedTab[13].OpIndex := 28;
336   RedTab[14].OpSymbol := character[ceiling];
337   RedTab[14].OpIndex := 29; RedTab[15].OpSymbol := character[floor];
338   RedTab[15].OpIndex := 30;
339   RedTab[16].OpSymbol := character[largeCircle];
340   RedTab[16].OpIndex := 31; digits[OneSymbol] := 1;
341   digits[TwoSymbol] := 2; digits[ThreeSymbol] := 3;
342   digits[FourSymbol] := 4; digits[FiveSymbol] := 5;
343   digits[SixSymbol] := 6; digits[SevenSymbol] := 7;
344   digits[EightSymbol] := 8; digits[NineSymbol] := 9;
345   digits[ZeroSymbol] := 0;
346 end { filluptables };
347
348 procedure PrintAPLStatement;
349
350 var
351   prefix, num: integer;
352   index: integer;
353
354 begin
355   for index := 1 to LineLength do
356     begin
357       if APLstatement[index] > 6000
358       then
359         begin
360           prefix := APLstatement[index] div 100; write(chr(prefix));
361           num := APLstatement[index] - 100 * prefix;
362           write(chr(num));
363         end
364       else write(chr(APLstatement[index]))
365       end;
366     end
367   writeln
368 end { printaplstatement };
369
370 procedure SError(ErrorIndex: integer);
371
372 var
373   MsgCol: integer;
374
375 begin

```

```

371 TokenError := true;
372 for MsgCol := 1 to MessageLength do
373   write(ErrorMsgs[ErrorIndex, MsgCol]);
374 writeln; PrintAPLstatement ( echo statement to user );
375 for MsgCol := 1 to (position - 1) do write(' ');
376 writeln(chr(character[UpArrow])) ( print pointer to user error );
377 end ( error );
378
379
380 procedure SkipSpaces;
381
382 begin
383   while (APLstatement[position] = character[space]) and (position <=
384     LineLength) do
385     position := position + 1
386   end ( skipspaces );
387
388
389 procedure GetAPLstatement;
390
391 var
392   InputChar: char;
393   TestForPrefix: integer;
394   FirstTry: Boolean;
395
396 begin
397   for LineLength := 1 to MaxInputLine do
398     APLstatement[LineLength] := character[space] ( blank out line );
399   LineLength := 0; FirstTry := true; position := 1;
400   LineTooLong := false;
401   APLstatement[InputArraySize] := character[omega];
402   APLstatement[InputArraySize - 1] := character[space]
403   ( set end-of-line );
404   repeat
405     begin
406       if not FirstTry then getseg(input) ( test for "or" only );
407       FirstTry := false;
408       while (not eoln(input)) and (not LineTooLong) do
409         if LineLength < MaxInputLine
410         then
411           begin
412             LineLength := LineLength + 1; read(InputChar);
413
414             ( The following code would be removed for non-CDC installations )
415             TestForPrefix := ord(InputChar);
416             if (TestForPrefix = prefix1) or (TestForPrefix = prefix2)
417             then
418               begin
419                 read(InputChar);
420                 APLstatement[LineLength] := 100 * TestForPrefix + ord(
421                   InputChar);
422               end
423             else
424
425             (
426
427               APLstatement[LineLength] := ord(InputChar)
428             end
429             else LineTooLong := true
430           end
431         until LineLength <> 0 ( reject null lines );
432         if LineTooLong then SError(71)
433         end ( getaplstatement );
434
435 function ItsADigit(TestChar: integer): Boolean;
436
437 var
438   DigitIndex: APLcharSet;
439
440 begin ( test to see if input character is a digit )
441   ItsADigit := false;
442   for DigitIndex := OneSymbol to ZeroSymbol do
443     if TestChar = character[DigitIndex] then ItsADigit := true
444   end ( itsadigit );
445
446
447 function ItsALetter(TestChar: integer): Boolean;
448
449 var
450   LetterIndex: APLcharSet;
451
452 begin ( test to see if input character is a letter )
453   ItsALetter := false;
454   for LetterIndex := asymbol to Zsymbol do
455     if TestChar = character[LetterIndex] then ItsALetter := true
456   end ( itsaletter );
457
458
459 function CharToNum(TestChar: integer): integer;
460
461 var
462   DigitIndex: APLcharSet;
463
464 begin ( change a character to a number )
465   for DigitIndex := OneSymbol to ZeroSymbol do
466     if TestChar = character[DigitIndex]
467     then CharToNum := digits[DigitIndex]
468   end ( chartonum );
469
470
471 function NamesMatch(NameOne, NameToo: PackedString): Boolean;
472
473 var
474   Index: integer;
475
476 begin ( see if two names (identifiers) are the same )
477   NamesMatch := true;
478   for index := 1 to MaxVarNameLength do
479     if NameOne[index] <> NameToo[index] then NamesMatch := false
480   end ( namesmatch );
481
482
483 procedure TableLookUp(TestChar, TableLength: integer; table: OpTable;
484   var TableIndex: integer);
485
486 var
487   Index: integer;
488
489 begin ( check for membership in a given table )
490   TableIndex := 0;
491   for index := 1 to TableLength do
492     if TestChar = table[index].OpSymbol then TableIndex := index
493   end ( tablelookup );
494
495
496 procedure identifier(var name: PackedString; var ItsAnIdentifier:
497   Boolean);
498
499 var
500   NameLength: integer;
501   NameTooLong: Boolean;
502
503 begin
504   ItsAnIdentifier := false; SkipSpaces;
505   if ItsALetter(APLstatement[position])
506   then
507     begin
508       NameTooLong := false; ItsAnIdentifier := true;
509       for NameLength := 1 to MaxVarNameLength do ( blank out name )
510         name[NameLength] := character[space];
511       NameLength := 0;
512       while (ItsALetter(APLstatement[position]) or (ItsADigit(
513         APLstatement[position])) do
514         begin ( build identifier )
515           NameLength := NameLength + 1;
516           if NameLength <= MaxVarNameLength
517           then name[NameLength] := APLstatement[position]
518           else NameTooLong := true;
519           position := position + 1
520         end;
521       if NameTooLong
522       then SError(70) ( name greater than maxlength )
523       end
524     end ( identifier );
525
526
527 procedure MakeNumber(var RealNumber: real; var ItsANumber: Boolean);
528
529 var
530   sign, DigitCount: integer;
531
532 begin ( convert character input string to numerical representation )
533   ItsANumber := false; SkipSpaces; sign := 1; DigitCount := 0;
534   RealNumber := 0.0;
535   if (APLstatement[position] = character[negative]) or (ItsADigit(
536     APLstatement[position]))
537   then
538     begin
539       ItsANumber := true;
540       if APLstatement[position] = character[negative]
541       then begin sign := - 1; position := position + 1 end;
542       if not ItsADigit(APLstatement[position])
543       then
544         begin
545           SError(1) ( digit must follow a minus sign );
546           ItsANumber := false;
547         end
548       else
549         begin ( form whole number portion )
550           while ItsADigit(APLstatement[position]) do
551             begin
552               RealNumber := 10.0 * RealNumber + CharToNum(APLstatement
553                 [position]);
554               position := position + 1
555             end;
556           if APLstatement[position] = character[period]
557           then
558             begin
559               position := position + 1;
560               while ItsADigit(APLstatement[position]) do
561                 begin ( form fractional portion )
562                   RealNumber := RealNumber + CharToNum(APLstatement[
563                     position]) * exp((- 1.0 - DigitCount) * 2.3025851
564                     );
565                   DigitCount := DigitCount + 1;
566                   position := position + 1;
567                 end;
568               if DigitCount = 0 then
569                 begin
570                   SError(2) ( digits must follow a decimal point );
571                   ItsANumber := false;
572                 end
573             end;
574           end;

```

```

575     RealNumber := RealNumber * sign
576     end
577   end
578   end { makeanumber };
579
580 function MonadicReference: Boolean;
581
582   var
583     SubPosition, TableIndex: integer;
584
585   begin { see if operator is monadic within context of input line }
586     MonadicReference := false;
587     if NewTokenPtr^.NextToken^.noun = StatEnd
588     then MonadicReference := true
589     else
590       begin
591         SubPosition := position - 1;
592         while (SubPosition > 0) and (APLstatement[SubPosition] =
593         character[space]) do
594           SubPosition := SubPosition - 1 { get last non-blank };
595         if SubPosition <> 0 then
596           TableLookup(APLstatement[SubPosition], 6, SpecTab, TableIndex)
597         ;
598         if (TableIndex <> 0) or (SubPosition = 0)
599         then MonadicReference := true
600         else
601           if (NewTokenPtr^.NextToken^.noun <> FormRes) and (NewTokenPtr
602           ^.NextToken^.noun <> FormArg) and (NewTokenPtr^.NextToken
603           ^.noun <> GlobVar) and (NewTokenPtr^.NextToken^.noun <>
604           constant) and (APLstatement[SubPosition] <> character[
605           period]) and (APLstatement[SubPosition] <> character[
606           RightParen]) and (APLstatement[SubPosition] <> character[
607           RightBracket])
608           then MonadicReference := true
609         end
610       end { monadicreference };
611     end
612
613 procedure DyadicOpCheck;
614
615   var
616     TableIndex: integer;
617
618   begin
619     TableLookup(APLstatement[position], 16, DOpTab, TableIndex);
620     if TableIndex = 0
621     then
622       begin
623         TableLookup(APLstatement[position], 12, CharTab, TableIndex);
624         if TableIndex = 0
625         then
626           if APLstatement[position] = character[SouthCap]
627           then
628             begin
629               OldTokenPtr := SaveTokenPtr; dispose(NewTokenPtr);
630               NewTokenPtr := SaveTokenPtr; position := LineLength + 1;
631               end { this was a comment - ignore remainder of line }
632             else SError(4) { invalid character encountered }
633             else
634               begin { special character encountered }
635                 NewTokenPtr^.noun := SpecOper;
636                 NewTokenPtr^.CharIndx := TableIndex
637               end
638             end
639           else
640             if MonadicReference
641             then SError(74) { monadic reference to dyadic operator }
642             else
643               begin { operator is dyadic }
644                 NewTokenPtr^.noun := DyadOper;
645                 NewTokenPtr^.DOpIndx := TableIndex
646               end
647             end { dyadicopcheck };
648           end
649
650 procedure CheckOtherTables;
651
652   var
653     TableIndex: integer;
654     ChkIndex: integer;
655
656   function NextNonBlank: integer;
657
658     begin
659       ChkIndex := position + 1;
660       while (ChkIndex < LineLength) and (APLstatement[ChkIndex] =
661       character[space]) do
662         ChkIndex := ChkIndex + 1;
663       NextNonBlank := APLstatement[ChkIndex];
664     end { nextnonblank };
665
666   begin { checkothertables }
667     if NextNonBlank = character[forwardSlash]
668     then
669       begin
670         TableLookup(APLstatement[position], 16, RedTab, TableIndex);
671         if TableIndex = 0
672         then SError(72) { invalid reduction operator }
673         else
674           if not MonadicReference
675           then SError(73) { dyadic reduction reference }
676           else
677             begin { operator is valid reduction operator }
678               NewTokenPtr^.noun := ReductOper;
679               NewTokenPtr^.RedIndx := TableIndex;
680             end;
681             position := ChkIndex + 1;
682           end
683         else
684           begin
685             TableLookup(APLstatement[position], 9, MOpTab, TableIndex);
686             if TableIndex = 0 then DyadicOpCheck
687             else
688               if not MonadicReference then DyadicOpCheck
689               else
690                 begin { operator is monadic }
691                   NewTokenPtr^.noun := MonadOper;
692                   NewTokenPtr^.MonIndex := TableIndex;
693                 end;
694                 position := position + 1;
695               end
696             end { checkothertables };
697           end
698
699 procedure TryToGetANumber;
700
701   var
702     NumberCount: integer;
703     RealNumber: real;
704     ItsANumber: Boolean;
705
706   begin
707     NumberCount := 0; MakeNumber(RealNumber, ItsANumber);
708     if not ItsANumber then CheckOtherTables
709     else
710       begin { store values in value table }
711         new(NewValTabLink);
712         NewValTabLink^.NextValTabLink := OldValTabLink;
713         OldValTabLink := NewValTabLink;
714         NewValTabLink^.ForwardOrder := true;
715         if FunctionMode then NewValTabLink^.IntermedResult := false
716         else NewValTabLink^.IntermedResult := true;
717         switch := true;
718         while ItsANumber do
719           begin
720             NumberCount := NumberCount + 1; new(NewValues);
721             if switch
722             then
723               begin
724                 switch := false;
725                 NewValTabLink^.FirstValue := NewValues
726               end
727             else NewValPtr^.NextValue := NewValues;
728             NewValues^.RealVal := RealNumber; NewValPtr := NewValues;
729             MakeNumber(RealNumber, ItsANumber)
730           end;
731           NewValues^.NextValue := nil;
732           if NumberCount > 1
733           then
734             begin
735               NewValTabLink^.dimensions := 1 { number is a vector };
736               new(NewDim); NewValTabLink^.FirstDimen := NewDim;
737               NewDim^.dimenlength := NumberCount;
738               NewDim^.NextDimen := nil
739             end
740           else
741             begin
742               NewValTabLink^.dimensions := 0 { number is a scalar };
743               NewValTabLink^.FirstDimen := nil
744             end;
745           end;
746           NewTokenPtr^.noun := constant;
747           NewTokenPtr^.ValTabPtr := NewValTabLink;
748         end
749       end { trytogetanumber };
750     end
751
752 function NameInVarTable(name: PackedString; var VarPointer:
753 VarTabPtrType; TestFuncPtr: PtrFuncTab): Boolean;
754
755   var
756     Found: Boolean;
757
758   begin
759     found := false; VarPointer := OldVarTabPtr;
760     while (VarPointer <> nil) and (not found) do
761       begin
762         if (NamesMatch(name, VarPointer^.VarName) and (VarPointer^.
763         FuncTabPtr = TestFuncPtr) { test for global var }
764         then found := true
765         else VarPointer := VarPointer^.NextVarTabPtr
766         end;
767       NameInVarTable := found;
768     end { nameinvartable };
769   end
770
771 procedure AddNameToVarTable(name: PackedString);
772
773   begin { new variable name encountered }
774     new(NewVarTabPtr); NewVarTabPtr^.NextVarTabPtr := OldVarTabPtr;
775     OldVarTabPtr := NewVarTabPtr; NewVarTabPtr^.VarName := name;
776     NewVarTabPtr^.ValTabPtr := nil;
777   end

```

```

779 if NewTokenPtr <> nil
780 then
781   if (NewTokenPtr^.noun = FormRes) or (NewTokenPtr^.noun = FormArg
782     )
783     then NewVarTabPtr^.FuncTabPtr := NewFuncTabPtr
784     else NewVarTabPtr^.FuncTabPtr := nil
785   end { addnametovariable };
786
787
788 function FunctionAlreadyDefined(var NewFuncName: PackedString; var
789   FuncIndex: PtrFuncTab): Boolean;
790
791 var
792   found: Boolean;
793
794 begin
795   found := false; FuncIndex := OldFuncTabPtr;
796   while (FuncIndex <> nil) and (not found) and (NewFuncTabPtr <>
797     nil) do
798     if NamesMatch(FuncIndex^.FuncName, NewFuncName)
799       then found := true
800       else FuncIndex := FuncIndex^.NextFuncTabPtr;
801     FunctionAlreadyDefined := found
802   end { functionalreadydefined };
803
804
805 procedure MakeTokenLink;
806
807 begin
808   new(NewTokenPtr); NewTokenPtr^.NextToken := OldTokenPtr;
809   SaveTokenPtr := OldTokenPtr; OldTokenPtr := NewTokenPtr
810 end { maketokenlink };
811
812
813 procedure ProcessFunctionHeader;
814
815 var
816   DummyPtr: ^FuncTab;
817   name1, name2, name3: PackedString;
818   ItsAnIdentifier, FuncHeadError: Boolean;
819   ArityIndex: integer;
820
821 begin
822   FuncHeadError := false; FunctionMode := true;
823   FuncStatements := - 1;
824   if FirstFunction
825     then begin FuncStatements := 0; FirstFunction := false; end;
826   ArityIndex := 1; position := position + 1;
827   identifier(name1, ItsAnIdentifier);
828   if not ItsAnIdentifier
829     then
830       begin
831         SError(7) { unrecognizable function/argument name };
832         FunctionMode := false { exit function mode };
833         FuncHeadError := true
834       end
835     else
836       begin
837         new(NewFuncTabPtr); SkipSpaces;
838         if APLstatement[position] = character[LeftArrow]
839           then
840             begin
841               NewFuncTabPtr^.result := true { explicit result };
842               NewFuncTabPtr^.ResultName := name1;
843               position := position + 1;
844               identifier(name1, ItsAnIdentifier);
845               if not ItsAnIdentifier then
846                 begin
847                   SError(6)
848                   { unrecognizable name to right of explicit res };
849                   FuncHeadError := true
850                 end
851             else NewFuncTabPtr^.result := false { no explicit result };
852             SkipSpaces;
853             if (position <= LineLength) and (not FuncHeadError)
854               then
855                 begin
856                   identifier(name2, ItsAnIdentifier);
857                   if not ItsAnIdentifier
858                     then
859                       begin
860                         SError(7) { invalid function/argument name };
861                         FuncHeadError := true
862                       end
863                     else ArityIndex := 2
864                   end;
865                 SkipSpaces;
866                 if (position <= LineLength) and (not FuncHeadError)
867                   then
868                     begin
869                       identifier(name3, ItsAnIdentifier);
870                       if not ItsAnIdentifier
871                         then
872                           begin
873                             SError(9) { invalid function right argument name };
874                             FuncHeadError := true
875                           end
876                         else ArityIndex := 3
877                       end;
878                     SkipSpaces;
879

```

```

880 if (position <= LineLength) and (not FuncHeadError) then
881   begin
882     SError(3)
883     { extraneous characters to right of function header };
884     FuncHeadError := true
885   end;
886   case ArityIndex of
887     1:
888       begin
889         NewFuncTabPtr^.arity := niladic;
890         NewFuncTabPtr^.FuncName := name1;
891       end;
892     2:
893       begin
894         NewFuncTabPtr^.arity := monadic;
895         NewFuncTabPtr^.FuncName := name1;
896         NewFuncTabPtr^.RightArg := name2;
897         AddNameToVarTable(name2);
898         NewVarTabPtr^.FuncTabPtr := NewFuncTabPtr;
899       end;
900     3:
901       begin
902         NewFuncTabPtr^.arity := dyadic;
903         NewFuncTabPtr^.LeftArg := name1;
904         NewFuncTabPtr^.FuncName := name2;
905         NewFuncTabPtr^.RightArg := name3;
906         AddNameToVarTable(name1);
907         NewVarTabPtr^.FuncTabPtr := NewFuncTabPtr;
908         AddNameToVarTable(name3);
909         NewVarTabPtr^.FuncTabPtr := NewFuncTabPtr;
910       end
911   end { case };
912   if FunctionAlreadyDefined(NewFuncTabPtr^.FuncName, DummyPtr)
913     then
914       begin
915         SError(5) { function already defined };
916         FuncHeadError := true;
917       end;
918     if FuncHeadError then
919       begin
920         dispose(NewFuncTabPtr) { header no good };
921         FunctionMode := false { exit function mode };
922         NewFuncTabPtr := OldFuncTabPtr;
923       end
924   end { processfuncheader };
925
926
927
928 procedure DestroyStatement;
929
930 var
931   DumTokenPtr: ^TokenTable;
932   AuxSubrTabPtr: ^SubrTab;
933
934 begin
935   if SubrTabPtr <> nil
936     then
937       begin
938         while SubrTabPtr^.LastSubrPtr <> nil do
939           begin
940             AuxSubrTabPtr := SubrTabPtr;
941             SubrTabPtr := SubrTabPtr^.LastSubrPtr;
942             dispose(AuxSubrTabPtr);
943           end;
944         dispose(SubrTabPtr);
945       end;
946   DumTokenPtr := OldTokenPtr;
947   while DumTokenPtr <> HoldTokenPtr do
948     begin
949       OldTokenPtr := OldTokenPtr^.NextToken; dispose(DumTokenPtr);
950       DumTokenPtr := OldTokenPtr
951     end;
952   NewTokenPtr := HoldTokenPtr;
953   OldTokenPtr := HoldTokenPtr
954   { return pointer to end of last good line }
955 end { destroystatement };
956
957
958 procedure ReverseLinkList(var ArgPtr: TypeValTabPtr);
959
960 var
961   hold, TempPtr: ^values;
962
963 begin { reverselinklist }
964   ValPtr := ArgPtr^.FirstValue; TempPtr := ValPtr^.NextValue;
965   while TempPtr <> nil do
966     begin
967       hold := TempPtr^.NextValue; TempPtr^.NextValue := ValPtr;
968       ValPtr := TempPtr; TempPtr := hold
969     end;
970   ArgPtr^.FirstValue^.NextValue := nil;
971   ArgPtr^.FirstValue := ValPtr;
972   if ArgPtr^.ForwardOrder
973     then ArgPtr^.ForwardOrder := false
974     else ArgPtr^.ForwardOrder := true { toggle list order switch }
975   end { reverselinklist };
976
977
978 procedure parser(var TokenTabPtr: TokenPtr; var PtrToDa: TypeValTabPtr);
979
980 var

```

```

981 VFuncHold: "vfunc { hold while searching };
982 AuxOperTabPtr: "OperandTab;
983 AuxSubrTabPtr: "SubrTab;
984 AuxRParamPtr: "RParamTab;
985 AuxLParamPtr: "LParamTab;
986 ValidExp: Boolean { true if valid expression };
987 cnt: integer;
988 npv: integer { number of indices };
989 assign, assign1: Boolean { assign.in progress };
990 DoneSuccessor: Boolean;
991 DoneParse: Boolean;
992
993
994 procedure error(ErrorIndex: integer);
995
996 var
997   MsgCol: integer;
998
999 begin
1000   write(' ', ErrorIndex, ' ');
1001   for MsgCol := 1 to MessageLength do
1002     write(ErrorMsgs[ErrorIndex, MsgCol]);
1003   writeln; goto 100 { return to scanner };
1004 end { error };
1005
1006
1007 procedure release;
1008
1009 begin { releaseopertab }
1010   OperTabPtr := PtrLastOper;
1011   while OperTabPtr.LastOper <> nil do
1012     begin
1013       AuxOperTabPtr := OperTabPtr;
1014       OperTabPtr := OperTabPtr.LastOper; dispose(AuxOperTabPtr);
1015     end;
1016   end { releaseopertab };
1017
1018
1019 procedure expression(var ValidExp: Boolean);
1020   forward;
1021
1022
1023 procedure ReturnToCallingSubr;
1024
1025 var
1026   NamePtr: "VarTab;
1027
1028 begin { returntocallingsubr }
1029   if SubrTabPtr.CalledSubr.result
1030   then
1031     begin { place explicit result in opertab }
1032       if not NameInVarTable(SubrTabPtr.CalledSubr.ResultName,
1033         NamePtr, SubrTabPtr.CalledSubr)
1034       then error(11) { 'symbol not found' };
1035     else
1036       begin
1037         AuxOperTabPtr := OperTabPtr; new(OperTabPtr);
1038         OperTabPtr.LastOper := AuxOperTabPtr;
1039         PtrLastOper := OperTabPtr;
1040         OperTabPtr.OperPtr := NamePtr.ValTabPtr;
1041       end;
1042     end;
1043   { return to calling function }
1044   VFuncPtr := SubrTabPtr.StateCallingSubr;
1045   TokenTabPtr := SubrTabPtr.TokenCallingSubr.NextToken;
1046   if SubrTabPtr.CalledSubr.arity <> niladic
1047   then
1048     begin { monadic or dyadic }
1049       AuxRParamPtr := RParamPtr; RParamPtr := RParamPtr.LastParam;
1050       dispose(AuxRParamPtr);
1051       if SubrTabPtr.CalledSubr.arity = dyadic then
1052         begin { dyadic only }
1053           AuxLParamPtr := LParamPtr;
1054           LParamPtr := LParamPtr.LastParam; dispose(AuxLParamPtr);
1055         end;
1056       end;
1057       AuxSubrTabPtr := SubrTabPtr;
1058       SubrTabPtr := SubrTabPtr.LastSubrPtr; dispose(AuxSubrTabPtr);
1059     end { returntocallingsubr };
1060
1061
1062 function SpecSymbol(sym: integer): Boolean;
1063
1064 var
1065   ValidSym: Boolean;
1066
1067 begin { specsymbol }
1068   ValidSym := false;
1069   if TokenTabPtr.noun = SpecOper
1070   then
1071     if TokenTabPtr.CharIndx = sym then
1072       begin
1073         hold := TokenTabPtr;
1074         TokenTabPtr := TokenTabPtr.NextToken; ValidSym := true;
1075       end;
1076     SpecSymbol := ValidSym;
1077   end { specsymbol };
1078
1079
1080 procedure CallSubr;
1081
1082 var
1083   PtrToVarTab: "VarTab;
1084
1085 begin { callsubr }
1086   if SubrTabPtr.CalledSubr.arity <> niladic
1087   then
1088     begin
1089       if not NameInVarTable(SubrTabPtr.CalledSubr.RightArg,
1090         PtrToVarTab, SubrTabPtr.CalledSubr)
1091       then error(32);
1092       if PtrToVarTab.FuncTabPtr <> SubrTabPtr.CalledSubr
1093       then error(32) { program logic error, variable name of };
1094       { function argument not found in symbol table }
1095       AuxRParamPtr := RParamPtr; new(RParamPtr);
1096       RParamPtr.LastParam := AuxRParamPtr;
1097       PtrToVarTab.DeferredValTabPtr := RParamPtr;
1098       if SubrTabPtr.CalledSubr.arity = dyadic
1099       then
1100         begin { if dyadic }
1101           if not NameInVarTable(SubrTabPtr.CalledSubr.LeftArg,
1102             PtrToVarTab, SubrTabPtr.CalledSubr)
1103           then error(33);
1104           if PtrToVarTab.FuncTabPtr <> SubrTabPtr.CalledSubr
1105           then error(33) { same as error(32) };
1106           AuxLParamPtr := LParamPtr; new(LParamPtr);
1107           LParamPtr.LastParam := AuxLParamPtr;
1108           PtrToVarTab.DeferredValTabPtr := LParamPtr;
1109           LParamPtr.PtrVal := OperTabPtr.OperPtr;
1110           AuxOperTabPtr := OperTabPtr;
1111           OperTabPtr := OperTabPtr.LastOper;
1112           dispose(AuxOperTabPtr); PtrLastOper := OperTabPtr;
1113         end;
1114         RParamPtr.PtrVal := OperTabPtr.OperPtr;
1115         AuxOperTabPtr := OperTabPtr;
1116         OperTabPtr := OperTabPtr.LastOper; dispose(AuxOperTabPtr);
1117         PtrLastOper := OperTabPtr;
1118       end;
1119       TokenTabPtr := SubrTabPtr.CalledSubr.FirstStatement.NextStmnt;
1120       VFuncPtr := SubrTabPtr.CalledSubr.FirstStatement;
1121     end { callsubr };
1122
1123
1124 function FuncCall: Boolean;
1125
1126 var
1127   PtrToFuncTab: "FuncTab;
1128   NameOfFunc: PackedString;
1129   ValidFn: Boolean;
1130
1131 begin { funcall }
1132   ValidFn := false;
1133   if TokenTabPtr.noun = GlobVar
1134   then
1135     begin
1136       NameOfFunc := TokenTabPtr.VarTabPtr.VarName;
1137       if FunctionAlreadyDefined(NameOfFunc, PtrToFuncTab)
1138       then
1139         begin
1140           AuxSubrTabPtr := SubrTabPtr; new(SubrTabPtr);
1141           SubrTabPtr.LastSubrPtr := AuxSubrTabPtr;
1142           SubrTabPtr.CalledSubr := PtrToFuncTab;
1143           SubrTabPtr.TokenCallingSubr := TokenTabPtr;
1144           SubrTabPtr.StateCallingSubr := VFuncPtr;
1145           hold := TokenTabPtr;
1146           TokenTabPtr := TokenTabPtr.NextToken; ValidFn := true;
1147         end;
1148       end;
1149       FuncCall := ValidFn;
1150     end { funcall };
1151
1152
1153 procedure NumWrite(RealNo: real);
1154
1155 var
1156   prefix, root: integer;
1157   SigDig, ColCnt: integer;
1158
1159 begin { output a number }
1160   if RealNo >= 0.0
1161   then write(' ', RealNo: 12: 2) { output positive number };
1162   else
1163     begin { output negative number }
1164       RealNo := - 1.0 * RealNo;
1165       SigDig := trunc((ln(RealNo)) / (ln(10.0)));
1166       for ColCnt := 1 to (7 - SigDig) do write(' ');
1167       if character[negative] < 6000
1168       then write(chr(character[negative]));
1169     else
1170       begin
1171         prefix := character[negative] div 100;
1172         root := character[negative] - (100 * prefix);
1173         write(chr(prefix), chr(root));
1174       end;
1175       SigDig := SigDig + 5; write(RealNo: SigDig: 2);
1176     end { numwrite };
1177
1178
1179
1180 procedure OutPutVal;
1181

```

```

1182 var
1183   cnt: integer;
1184   AuxValuesPtr: ^values;
1185   DimHold, dimen1, dimen2, dimen3: integer;
1186   OutCnt1, OutCnt2, OutCnt3: integer;
1187   idimens: integer;
1188
1189 begin [ outputval ]
1190   cnt := 0; writeln; writeln;
1191   if not OperTabPtr^.OperPtr^.ForwardOrder
1192   then ReverseLinkList(OperTabPtr^.OperPtr);
1193   AuxValuesPtr := OperTabPtr^.OperPtr^.FirstValue;
1194   idimens := OperTabPtr^.OperPtr^.dimensions;
1195   if not (idimens in [0..3])
1196   then
1197     begin
1198       for ColCnt := 1 to MessageLength do
1199         write(ErrorMsgs[60], ColCnt);
1200       writeln;
1201     end
1202   else
1203     if AuxValuesPtr = nil
1204     then
1205       begin
1206         for ColCnt := 1 to MessageLength do
1207           write(ErrorMsgs[61], ColCnt);
1208         writeln;
1209       end
1210     else
1211       if idimens = 0
1212       then begin MunWrite(AuxValuesPtr^.RealVal); writeln; end
1213       else
1214         begin
1215           dimen1 := OperTabPtr^.OperPtr^.FirstDimen^.dimenlength;
1216           if idimens >= 2
1217           then
1218             dimen2 := OperTabPtr^.OperPtr^.FirstDimen^.NextDimen
1219               ^dimenlength
1220           else dimen2 := 1;
1221           if idimens = 3
1222           then
1223             dimen3 := OperTabPtr^.OperPtr^.FirstDimen^.NextDimen
1224               ^NextDimen^.dimenlength
1225           else dimen3 := 1;
1226           if idimens = 3 then
1227             begin [ rotate dimensions ]
1228               DimHold := dimen1; dimen1 := dimen2;
1229               dimen2 := dimen3; dimen3 := DimHold;
1230             end;
1231           for OutCnt3 := 1 to dimen3 do
1232             begin
1233               for OutCnt2 := 1 to dimen2 do
1234                 begin
1235                   for OutCnt1 := 1 to dimen1 do
1236                     begin
1237                       cnt := cnt + 1;
1238                       if ((cnt - 1) mod 5) = 0 and (cnt <> 1)
1239                       then begin writeln; write(' '); end;
1240                       MunWrite(AuxValuesPtr^.RealVal);
1241                       AuxValuesPtr := AuxValuesPtr^.NextValue;
1242                     end;
1243                   if idimens >= 2
1244                   then begin writeln; cnt := 0; end;
1245                 end;
1246               writeln; writeln;
1247             end;
1248           (writeln; )
1249         end;
1250       end [ outputval ];
1251
1252 function variable: Boolean;
1253
1254 var
1255   globOrDummy: Boolean ( gord );
1256   PassedAdj: ^VarTab ( k );
1257   rarg: Boolean ( rd );
1258   ParmPtr: ^ValTab ( pt );
1259   ValidVar: Boolean;
1260   ValidIndex: Boolean;
1261
1262 procedure InputVal;
1263
1264 var
1265   AuxPtrToDa: ^ValTab;
1266   AuxValuesPtr: ^values;
1267   Aux2ValuesPtr: ^values;
1268   RealV: real;
1269   boolv: Boolean;
1270   cntnr, cnt: integer;
1271   AuxDimenFoPtr: ^DimenInfo;
1272
1273 begin [ inputval ]
1274   cnt := 0; position := 1; AuxPtrToDa := PtrToDa;
1275   new(PtrToDa); AuxPtrToDa^.NextValTabLink := PtrToDa;
1276   AuxOperTabPtr := OperTabPtr; new(OperTabPtr);
1277   PtrLastOper := OperTabPtr;
1278   OperTabPtr^.LastOper := AuxOperTabPtr;
1279   OperTabPtr^.OperPtr := PtrToDa; new(Aux2ValuesPtr);
1280   PtrToDa^.FirstValue := Aux2ValuesPtr;

```

```

1281   for cntnr := 1 to MessageLength do write(ErrorMsgs[63], cntnr);
1282   writeln; readln; GetAPLstatement;
1283   repeat
1284     MakeNumber(RealV, boolv); SkipSpaces;
1285     if not boolv
1286     then
1287       begin
1288         for ColCnt := 1 to MessageLength do
1289           write(ErrorMsgs[62], ColCnt);
1290         writeln; position := 1; cnt := 0;
1291         Aux2ValuesPtr := OperTabPtr^.OperPtr^.FirstValue;
1292         for cntnr := 1 to MessageLength do
1293           write(ErrorMsgs[63], cntnr);
1294         writeln; readln; GetAPLstatement
1295       end
1296     else
1297       begin
1298         cnt := cnt + 1; AuxValuesPtr := Aux2ValuesPtr;
1299         new(Aux2ValuesPtr); AuxValuesPtr^.RealVal := RealV;
1300         AuxValuesPtr^.NextValue := Aux2ValuesPtr;
1301       end;
1302     until position > LineLength;
1303     dispose(Aux2ValuesPtr); AuxValuesPtr^.NextValue := nil;
1304     PtrToDa^.IntermedResult := false; PtrToDa^.dimensions := 1;
1305     PtrToDa^.ForwardOrder := true;
1306     PtrToDa^.NextValTabLink := nil; new(AuxDimenFoPtr);
1307     PtrToDa^.FirstDimen := AuxDimenFoPtr;
1308     AuxDimenFoPtr^.dimenlength := cnt;
1309     AuxDimenFoPtr^.NextDimen := nil;
1310   end [ inputval ];
1311
1312 procedure GetArrayPosition(var ValuesPtr: TypeValuesPtr);
1313
1314 var
1315   indice: real;
1316   kcnc: integer;
1317   sl: integer;
1318   AuxDimenFoPtr: ^DimenInfo;
1319
1320 begin [ getarrayposition ]
1321   if npv <> ParmPtr^.dimensions then error(35);
1322   ( 'wrong num. of subscripts' )
1323   sl := 0; AuxOperTabPtr := OperTabPtr;
1324   AuxDimenFoPtr := ParmPtr^.FirstDimen;
1325   for kcnc := 1 to npv do
1326     begin
1327       if AuxOperTabPtr^.OperPtr^.dimensions <> 0
1328       then error(35) ( 'non-scalar indices' );
1329       indice := AuxOperTabPtr^.OperPtr^.FirstValue^.RealVal;
1330       if indice - 1.0 * trunc(indice) <> 0.0
1331       then error(37) ( 'non-integer indices' );
1332       if not (trunc(indice) in [1..AuxDimenFoPtr^.dimenlength
1333         ])
1334       then error(38) ( 'out of range index' );
1335       sl := (sl * AuxDimenFoPtr^.dimenlength) + trunc(indice) -
1336         1;
1337       AuxOperTabPtr := AuxOperTabPtr^.LastOper;
1338       dispose(OperTabPtr); OperTabPtr := AuxOperTabPtr;
1339       AuxDimenFoPtr := AuxDimenFoPtr^.NextDimen;
1340     end;
1341   ValuesPtr := ParmPtr^.FirstValue;
1342   while sl <> 0 do { determine which value in }
1343     [ pt[sval(sv)][sval(sv-1)]...[sval(sv-npv+1)] ]
1344     := sval(sv-npv)
1345     begin ValuesPtr := ValuesPtr^.NextValue; sl := sl - 1; end;
1346   end [ getarrayposition ];
1347
1348 procedure LinkResults;
1349
1350 var
1351   PtrToValues: ^values;
1352
1353 begin [ linkresults ]
1354   if npv = 0
1355   then
1356     begin
1357       if not globOrDummy
1358       then
1359         if rarg then RParmPtr^.PtrVal := OperTabPtr^.OperPtr
1360         else LParmPtr^.PtrVal := OperTabPtr^.OperPtr
1361         else PassedAdj^.ValTabPtr := OperTabPtr^.OperPtr
1362       end
1363     else
1364       begin
1365         if globOrDummy then ParmPtr := PassedAdj^.ValTabPtr
1366         else ParmPtr := PassedAdj^.DeferredValTabPtr^.PtrVal;
1367         GetArrayPosition(PtrToValues);
1368         if OperTabPtr^.OperPtr^.dimensions <> 0
1369         then error(36) ( 'assigned expression not a scalar' );
1370         PtrToValues^.RealVal := OperTabPtr^.OperPtr^.FirstValue
1371           ^RealVal;
1372       end;
1373     AuxOperTabPtr := OperTabPtr;
1374     OperTabPtr := OperTabPtr^.LastOper; dispose(AuxOperTabPtr);
1375     PtrLastOper := OperTabPtr;
1376   end [ linkresults ];
1377
1378 procedure StackPointers;

```

```

1384 var
1385   AuxPtrToDa: ^ValTab;
1386   PtrToValues, AuxValuesPtr: ^values;
1387
1388 begin ( stackpointers )
1389   ;
1390   if npv = 0
1391   then
1392     begin
1393       AuxOperTabPtr := OperTabPtr; new(OperTabPtr);
1394       OperTabPtr^.LastOper := AuxOperTabPtr;
1395       OperTabPtr^.OperPtr := ParmPtr;
1396       PtrLastOper := OperTabPtr
1397     end
1398   else
1399     begin
1400       AuxPtrToDa := PtrToDa; new(PtrToDa);
1401       PtrToDa^.NextValTabLink := AuxPtrToDa;
1402       PtrToDa^.IntermedResult := true;
1403       PtrToDa^.dimensions := 0; PtrToDa^.FirstDimen := nil;
1404       PtrToDa^.ForwardOrder := true; new(AuxValuesPtr);
1405       PtrToDa^.FirstValue := AuxValuesPtr;
1406       GetArrayPosition(PtrToValues);
1407       PtrToDa^.FirstValue^.RealVal := PtrToValues^.RealVal;
1408       PtrToDa^.FirstValue^.NextValue := nil;
1409       AuxOperTabPtr := OperTabPtr; new(OperTabPtr);
1410       OperTabPtr^.LastOper := AuxOperTabPtr;
1411       OperTabPtr^.OperPtr := PtrToDa;
1412       PtrLastOper := OperTabPtr;
1413     end;
1414   end ( stackpointers );
1415
1416 function SimpleVariable: Boolean;
1417
1418 var
1419   ValidSv: Boolean;
1420
1421 begin ( simplevariable )
1422   ValidSv := false; rarg := false; globOrDummy := false;
1423   if assign
1424   then
1425     begin
1426       if (TokenTabPtr^.noun = FormRes) or (TokenTabPtr^.noun =
1427         GlobVar)
1428       then
1429         begin
1430           globOrDummy := true;
1431           PassedAdj := TokenTabPtr^.VarTabPtr;
1432           hold := TokenTabPtr;
1433           TokenTabPtr := TokenTabPtr^.NextToken;
1434           ValidSv := true
1435         end
1436       else
1437         if TokenTabPtr^.noun = FormArg
1438         then
1439           begin
1440             if NamesMatch(TokenTabPtr^.VarTabPtr^.FuncTabPtr^.
1441               LeftArg, TokenTabPtr^.VarTabPtr^.VarName)
1442             then rarg := true;
1443             PassedAdj := TokenTabPtr^.VarTabPtr
1444           end
1445         end
1446       end
1447     else
1448       begin
1449         if (TokenTabPtr^.noun = FormRes) or (TokenTabPtr^.noun =
1450           GlobVar)
1451         then
1452           begin
1453             ParmPtr := TokenTabPtr^.VarTabPtr^.ValTabPtr;
1454             if ParmPtr <> nil then
1455               begin
1456                 hold := TokenTabPtr;
1457                 TokenTabPtr := TokenTabPtr^.NextToken;
1458                 ValidSv := true
1459               end
1460             end
1461           else
1462             begin
1463               if TokenTabPtr^.noun = FormArg
1464               then
1465                 begin
1466                   if NamesMatch(TokenTabPtr^.VarTabPtr^.FuncTabPtr^.
1467                     LeftArg, TokenTabPtr^.VarTabPtr^.VarName)
1468                   then ParmPtr := LParmPtr^.PtrVal;
1469                   else ParmPtr := RParmPtr^.PtrVal;
1470                   hold := TokenTabPtr;
1471                   TokenTabPtr := TokenTabPtr^.NextToken;
1472                   ValidSv := true;
1473                 end;
1474               end;
1475             end;
1476           SimpleVariable := ValidSv;
1477         end ( simple variable );
1478
1479 procedure index(var ValidI: Boolean);
1480
1481 var
1482   ValidE1, ValidE2: Boolean;
1483
1484   begin ( index )
1485     ValidI := false; expression(ValidE1);
1486     if ValidE1
1487     then
1488       begin
1489         npv := 1 ( no. of index expressions );
1490         while SpecSymbol(XSemicolon) do
1491           begin
1492             npv := npv + 1; expression(ValidE2);
1493             if not ValidE2 then error(39);
1494             ( 'invalid index expression' )
1495           end;
1496           ValidI := true;
1497         end;
1498       end ( index );
1499
1500   begin ( variable )
1501     ValidVar := false; npv := 0;
1502     if not assign
1503     then
1504       if SpecSymbol(XQuadSym)
1505       then begin InputVal; ValidVar := true end
1506       else
1507         begin
1508           if SpecSymbol(XRightBracket)
1509           then
1510             begin
1511               index(ValidIndex);
1512               if (not ValidIndex) or (not SpecSymbol(XLeftBracket))
1513               then error(34) ( 'invalid index expression' );
1514             end;
1515             if SimpleVariable
1516             then begin StackPointers; ValidVar := true end
1517           end
1518         else
1519           if SpecSymbol(XQuadSym)
1520           then begin OutPutVal; ValidVar := true end
1521           else
1522             begin
1523               if SpecSymbol(XRightBracket)
1524               then
1525                 begin
1526                   index(ValidIndex);
1527                   if (not ValidIndex) or (not SpecSymbol(XLeftBracket))
1528                   then error(34) ( 'invalid index expression' );
1529                 end;
1530                 if SimpleVariable
1531                 then begin LinkResults; ValidVar := true; end;
1532               end;
1533             end;
1534             variable := ValidVar;
1535           end ( variable );
1536
1537 procedure primary(var valid: Boolean) ( recursive entry );
1538
1539 var
1540   ValidX: Boolean;
1541   assign: Boolean;
1542
1543 function vector: Boolean;
1544
1545 var
1546   vec: Boolean;
1547
1548   begin ( vector )
1549     vec := false;
1550     if TokenTabPtr^.noun = constant
1551     then
1552       begin
1553         AuxOperTabPtr := OperTabPtr; new(OperTabPtr);
1554         PtrLastOper := OperTabPtr;
1555         OperTabPtr^.LastOper := AuxOperTabPtr;
1556         OperTabPtr^.OperPtr := TokenTabPtr^.ValTabPtr;
1557         hold := TokenTabPtr;
1558         TokenTabPtr := TokenTabPtr^.NextToken; vec := true;
1559       end;
1560     vector := vec;
1561   end ( vector );
1562
1563   begin ( primary )
1564     valid := true;
1565     if not vector
1566     then
1567       begin
1568         assign := false;
1569         if not variable
1570         then
1571           if SpecSymbol(XRightPar)
1572           then
1573             begin
1574               expression(ValidX);
1575               if not ValidX
1576               then error(14) ( 'non-valid exp within parens' );
1577             end;
1578           else
1579             if not SpecSymbol(XLeftPar)
1580             then
1581               error(15)
1582             ( 'right paren not balanced with left paren' );
1583             else valid := true
1584           end
1585         end
1586       end
1587     end

```

```

1588     end
1589     else
1590     if not FunctCall then valid := false
1591     else begin CallSubr; primary(valid); end;
1592 end;
1593 end ( primary );
1594
1595 procedure expression ( recursive );
1596
1597 var
1598 DoneExp, ValidPri, ValidFunc, ValidAssn: Boolean;
1599 code: integer;
1600
1601
1602
1603 procedure assignment(var valida: Boolean);
1604
1605 begin ( assignment )
1606 valida := false;
1607 if SpecSymbol(XLeftArrow)
1608 then
1609 begin
1610 assign := true; assign1 := true;
1611 if variable then valida := true
1612 else error(8) ( result of an assn not a valid variable );
1613 valida := true; assign := false;
1614 end;
1615 end ( assignment );
1616
1617
1618 function mop: Boolean;
1619
1620 var
1621 ValidM: Boolean;
1622
1623 begin ( mop )
1624 ValidM := false;
1625 if (TokenTabPtr^.noun = MonadOper) or (TokenTabPtr^.noun =
1626 ReductOper)
1627 then
1628 begin
1629 if TokenTabPtr^.noun = MonadOper
1630 then code := MOpTab[TokenTabPtr^.MonIndex].OpIndex
1631 else code := RedTab[TokenTabPtr^.RedIndex].OpIndex;
1632 hold := TokenTabPtr;
1633 TokenTabPtr := TokenTabPtr^.NextToken; ValidM := true;
1634 end;
1635 mop := ValidM;
1636 end ( mop );
1637
1638
1639 function dop: Boolean;
1640
1641 var
1642 ValidD: Boolean;
1643
1644 begin ( dop )
1645 ValidD := false;
1646 if TokenTabPtr^.noun = DyadOper
1647 then
1648 begin
1649 code := DOpTab[TokenTabPtr^.DOpIndex].OpIndex;
1650 hold := TokenTabPtr;
1651 TokenTabPtr := TokenTabPtr^.NextToken;
1652 if (code > 80) then ValidD := true
1653 else
1654 if TokenTabPtr^.noun = SpecOper
1655 then
1656 if SpecSymbol(XPeriod)
1657 then
1658 begin
1659 if TokenTabPtr^.noun = DyadOper
1660 then
1661 begin
1662 if DOpTab[TokenTabPtr^.DOpIndex].OpIndex <= 80
1663 then
1664 begin
1665 code := code + (100 * DOpTab[TokenTabPtr^.
1666 DOpIndex].OpIndex);
1667 hold := TokenTabPtr;
1668 TokenTabPtr := TokenTabPtr^.NextToken;
1669 ValidD := true
1670 end
1671 else error(27) ( 'invalid inner product exp' )
1672 end
1673 else
1674 if TokenTabPtr^.noun = SpecOper
1675 then
1676 begin
1677 if SpecSymbol(XLittleCircle)
1678 then
1679 begin code := 10 * code; ValidD := true
1680 end
1681 else error(26) ( 'inval outer prod exp' )
1682 end
1683 else error(26) ( same as above )
1684 end
1685 else ValidD := true
1686 else ValidD := true;
1687 end;
1688 dop := ValidD;

```

```

1689 end ( dop );
1690
1691 function ItsBoolean(test: real): Boolean;
1692
1693 begin
1694 if (test = 1.0) or (test = 0.0) then ItsBoolean := true
1695 else ItsBoolean := false
1696 end ( itsboolean );
1697
1698
1699 procedure DyadComp(var SFloat: real; value: real; code: integer);
1700 ( compute result of dyadic operation )
1701
1702 begin
1703 case code of
1704 ( left codes - reduction ops / right codes - dyadic ops )
1705 2, 52: SFloat := value + SFloat ( addition );
1706 3, 53: SFloat := value - SFloat ( subtraction );
1707 4, 54: SFloat := value * SFloat ( multiplication );
1708 5, 55:
1709 if SFloat = 0.0
1710 then error(20) ( attempted division by zero )
1711 else SFloat := value / SFloat ( division );
1712 6, 56:
1713 if value > 0.0
1714 then
1715 SFloat := exp(SFloat * ln(value))
1716 ( number raised to a power )
1717 else
1718 SFloat := 1.0 / (exp(SFloat * ln(abs(value))));
1719 21, 71:
1720 if value = SFloat ( equality ) then SFloat := 1.0
1721 else SFloat := 0.0;
1722 22, 72:
1723 if value <> SFloat ( inequality ) then SFloat := 1.0
1724 else SFloat := 0.0;
1725 23, 73:
1726 if value < SFloat ( less than ) then SFloat := 1.0
1727 else SFloat := 0.0;
1728 24, 74:
1729 if value <= SFloat ( less than or equal to )
1730 then SFloat := 1.0
1731 else SFloat := 0.0;
1732 25, 75:
1733 if value >= SFloat ( greater than or equal to )
1734 then SFloat := 1.0
1735 else SFloat := 0.0;
1736 26, 76:
1737 if value > SFloat ( greater than ) then SFloat := 1.0
1738 else SFloat := 0.0;
1739 27, 77:
1740 if (ItsBoolean(value)) and (ItsBoolean(SFloat))
1741 then
1742 if (value = 1.0) and (SFloat = 1.0) ( and )
1743 then SFloat := 1.0
1744 else SFloat := 0.0
1745 else error(19) ( value not boolean );
1746 28, 78:
1747 if (ItsBoolean(value)) and (ItsBoolean(SFloat))
1748 then
1749 if (value = 1.0) or (SFloat = 1.0) ( or )
1750 then SFloat := 1.0
1751 else SFloat := 0.0
1752 else error(19) ( value not boolean );
1753 29:
1754 if value > SFloat ( maximum or ceiling )
1755 then SFloat := value;
1756 30:
1757 if value < SFloat ( minimum or floor )
1758 then SFloat := value;
1759 31:
1760 if (value * SFloat) < 0.0
1761 then error(50) ( number and base of different sign )
1762 else
1763 SFloat := (ln(abs(SFloat))) / (ln(abs(value)))
1764 ( log to a base )
1765 end ( case )
1766 end ( dyadcomp );
1767
1768
1769 procedure IndexGenerator(arg: TypeValTabPtr);
1770 ( monadic iota operator )
1771
1772 var
1773 TotalIndex, TopValue: integer;
1774
1775 begin
1776 if arg^.dimensions <> 0
1777 then error(21) ( argument not a scalar )
1778 else
1779 if arg^.FirstValue^.RealVal < 0.0
1780 then error(22) ( argument is negative )
1781 else
1782 if (arg^.FirstValue^.RealVal) - (1.0 * trunc(arg^.
1783 FirstValue^.RealVal)) <> 0.0
1784 then error(23) ( argument is not an integer )
1785 else
1786 begin
1787 new(NewValTabLink);
1788 OldValTabLink^.NextValTabLink := NewValTabLink;

```

```

1789     NewValTabLink".NextValTabLink := nil;
1790     NewValTabLink".ForwardOrder := true;
1791     NewValTabLink".IntermedResult := true;
1792     NewValTabLink".dimensions := 1 ( result is a vector );
1793     new(NewDim);   NewValTabLink".FirstDimen := NewDim;
1794     TopValue := trunc(arg".FirstValue".RealVal)
1795     ( last index generd );
1796     NewDim".dimenlength := TopValue;
1797     NewDim".NextDimen := nil;   iotaIndex := 1;
1798     switch := true;
1799     while iotaIndex <= TopValue do
1800     begin
1801         new(NewValues);   NewValues".RealVal := iotaIndex;
1802         if switch
1803         then
1804             begin
1805                 switch := false;
1806                 NewValTabLink".FirstValue := NewValues
1807             end
1808             else NewValPtr".NextValue := NewValues;
1809                 NewValPtr := NewValues;
1810                 iotaIndex := iotaIndex + 1
1811             end;
1812         if switch
1813         then
1814             NewValTabLink".FirstValue := nil
1815             ( result is vector of length 0 )
1816             else NewValues".NextValue := nil
1817         end
1818     end ( indexgenerator );
1819
1820 procedure ravel(arg: TypeValTabPtr);
1821 { monadic comma operator }
1822
1823 var
1824     elements: integer;
1825
1826 begin
1827     new(NewValTabLink);
1828     OldValTabLink".NextValTabLink := NewValTabLink;
1829     NewValTabLink".NextValTabLink := nil;
1830     NewValTabLink".IntermedResult := true;
1831     NewValTabLink".ForwardOrder := arg".ForwardOrder;
1832     NewValTabLink".dimensions := 1 ( result is a vector );
1833     new(NewDim);   NewValTabLink".FirstDimen := NewDim;
1834     NewDim".NextDimen := nil;   switch := true;
1835     ValPtr := arg".FirstValue;   elements := 0;
1836     while ValPtr <> nil do
1837     begin ( duplicate values into result )
1838         new(NewValues);   NewValues".RealVal := ValPtr".RealVal;
1839         elements := elements + 1;
1840         if switch
1841         then
1842             begin
1843                 switch := false;
1844                 NewValTabLink".FirstValue := NewValues
1845             end
1846             else NewValPtr".NextValue := NewValues;
1847                 NewValPtr := NewValues;   ValPtr := ValPtr".NextValue
1848             end;
1849         NewDim".dimenlength := elements;
1850         if switch then NewValTabLink".FirstValue := nil
1851         else NewValues".NextValue := nil
1852     end ( ravel );
1853
1854 procedure ShapeOf(arg: TypeValTabPtr);
1855 { monadic rho operator }
1856
1857 begin
1858     new(NewValTabLink);
1859     OldValTabLink".NextValTabLink := NewValTabLink;
1860     NewValTabLink".NextValTabLink := nil;
1861     NewValTabLink".IntermedResult := true;
1862     NewValTabLink".ForwardOrder := true;
1863     NewValTabLink".dimensions := 1 ( result is a vector );
1864     new(NewDim);   NewDim".dimenlength := arg".dimensions;
1865     NewValTabLink".FirstDimen := NewDim;
1866     NewDim".NextDimen := nil;   switch := true;
1867     DimPtr := arg".FirstDimen;
1868     while DimPtr <> nil do
1869     begin ( argument dimensions become result values )
1870         new(NewValues);
1871         NewValues".RealVal := DimPtr".dimenlength;
1872         if switch
1873         then
1874             begin
1875                 switch := false;
1876                 NewValTabLink".FirstValue := NewValues
1877             end
1878             else NewValPtr".NextValue := NewValues;
1879                 NewValPtr := NewValues;   DimPtr := DimPtr".NextDimen
1880             end;
1881         if switch
1882         then
1883             NewValTabLink".FirstValue := nil
1884             ( result is a vector of length 0 )
1885         else NewValues".NextValue := nil
1886     end ( shapeof );
1887
1888

```

```

1891 procedure reduction(arg: TypeValTabPtr);
1892
1893 var
1894     counter, RowLength: integer;
1895     SFloat: real;
1896
1897 begin
1898     if (arg".dimensions = 0) or (arg".FirstValue = nil)
1899     then
1900         error(24) ( argument is a scalar or vector of length zero )
1901     else
1902         if (arg".dimensions = 1) and (arg".FirstDimen".dimenlength
1903         = 1)
1904         then error(51) ( argument is a vector of length one )
1905         else
1906             begin
1907                 new(NewValTabLink);
1908                 OldValTabLink".NextValTabLink := NewValTabLink;
1909                 NewValTabLink".NextValTabLink := nil;
1910                 NewValTabLink".IntermedResult := true;
1911                 if arg".ForwardOrder then ReverseLinkList(arg);
1912                 NewValTabLink".ForwardOrder := false;
1913                 NewValTabLink".dimensions := arg".dimensions - 1;
1914                 DimPtr := arg".FirstDimen;   switch := true;
1915                 while DimPtr".NextDimen <> nil do
1916                 begin ( build dimensions of result )
1917                     new(NewDim);
1918                     if switch
1919                     then
1920                         begin
1921                             switch := false;
1922                             NewValTabLink".FirstDimen := NewDim
1923                         end
1924                         else NewPtr".NextDimen := NewDim;
1925                             NewDim".dimenlength := DimPtr".dimenlength;
1926                             NewPtr := NewDim;   DimPtr := DimPtr".NextDimen
1927                         end;
1928                     if switch
1929                     then
1930                         NewValTabLink".FirstDimen := nil
1931                         ( arg is vector, result is scalar )
1932                     else NewDim".NextDimen := nil;
1933                         RowLength := DimPtr".dimenlength;
1934                         ValPtr := arg".FirstValue;   switch := true;
1935                         while ValPtr <> nil do
1936                         begin ( perform reduction )
1937                             SFloat := ValPtr".RealVal
1938                             ( sfloat gets last value in row );
1939                             ValPtr := ValPtr".NextValue;
1940                             for counter := 2 to RowLength do
1941                             begin
1942                                 DyadComp(SFloat, ValPtr".RealVal, code);
1943                                 ValPtr := ValPtr".NextValue
1944                             end;
1945                             new(NewValues);   NewValues".RealVal := SFloat;
1946                             if switch
1947                             then
1948                                 begin
1949                                     switch := false;
1950                                     NewValTabLink".FirstValue := NewValues
1951                                 end
1952                                 else NewValPtr".NextValue := NewValues;
1953                                     NewValPtr := NewValues
1954                                 end;
1955                             NewValues".NextValue := nil
1956                             end;
1957                         end ( reduction );
1958
1959 procedure monadic(arg: TypeValTabPtr; token: TokenPtr);
1960 { operations with codes between 1 and 31 }
1961
1962 begin
1963     if token".noun = ReductOper then reduction(arg)
1964     else
1965         if code > 20
1966         then
1967             case code of
1968                 21: IndexGenerator(arg);
1969                 22: ShapeOf(arg);
1970                 23: ravel(arg)
1971             end ( case )
1972         else
1973             begin
1974                 new(NewValTabLink);
1975                 OldValTabLink".NextValTabLink := NewValTabLink;
1976                 NewValTabLink".NextValTabLink := nil;
1977                 NewValTabLink".IntermedResult := true;
1978                 NewValTabLink".ForwardOrder := arg".ForwardOrder;
1979                 NewValTabLink".dimensions := arg".dimensions;
1980                 switch := true;   DimPtr := arg".FirstDimen;
1981                 while DimPtr <> nil do
1982                 begin ( duplicate dimensions of arg into result )
1983                     new(NewDim);
1984                     NewDim".dimenlength := DimPtr".dimenlength;
1985                     if switch
1986                     then
1987                         begin
1988                             switch := false;
1989                             NewValTabLink".FirstDimen := NewDim
1990                         end

```

```

1992     else NewPtr".NextDimen := NewDim;
1993     NewPtr := NewDim; DimPtr := DimPtr".NextDimen
1994   end;
1995   if switch
1996   then
1997     NewValTabLink".FirstDimen := nil ( result is a scalar )
1998   else NewDim".NextDimen := nil;
1999   switch := true; ValPtr := arg".FirstValue;
2000   while ValPtr <> nil do
2001     begin
2002       new(NewValues);
2003       if switch = true
2004       then
2005         begin
2006           switch := false;
2007           NewValTabLink".FirstValue := NewValues
2008         end
2009       else NewValPtr".NextValue := NewValues;
2010       NewValPtr := NewValues;
2011       case code of
2012         1:
2013           if ItsBoolean(ValPtr".RealVal)
2014             ( logical negation )
2015           then
2016             NewValues".RealVal := 1.0 - ValPtr".RealVal
2017           else error(19) ( value not boolean );
2018         2:
2019           NewValues".RealVal := ValPtr".RealVal
2020             ( no-op );
2021         3:
2022           NewValues".RealVal := 0.0 - ValPtr".RealVal
2023             ( negation );
2024         4:
2025           if ValPtr".RealVal > 0.0 ( signum )
2026           then NewValues".RealVal := 1.0
2027           else
2028             if ValPtr".RealVal < 0.0
2029             then NewValues".RealVal := - 1.0;
2030         5:
2031           if ValPtr".RealVal = 0.0 ( reciprocal )
2032           then error(54) ( attempted inverse of zero )
2033           else
2034             NewValues".RealVal := 1.0 / ValPtr".RealVal;
2035         6: NewValues".RealVal := exp(ValPtr".RealVal)
2036       end ( case );
2037       ValPtr := ValPtr".NextValue
2038     end;
2039     if switch then NewValTabLink".FirstValue := nil
2040     else NewValues".NextValue := nil
2041   end
2042 end ( monadic );
2043
2044 procedure catenate(LeftArg, RightArg: TypeValTabPtr);
2045 ( dyadic comma operator - joins 2 arguments )
2046
2047 var
2048   ResultLength: integer;
2049
2050 begin ( catenate )
2051   if (RightArg".dimensions > 1) or (LeftArg".dimensions > 1)
2052   then error(53) ( argument(s) with rank greater than 1 )
2053   else
2054     begin
2055       new(NewValTabLink);
2056       OldValTabLink".NextValTabLink := NewValTabLink;
2057       NewValTabLink".NextValTabLink := nil;
2058       NewValTabLink".IntermedResult := true;
2059       if not LeftArg".ForwardOrder
2060       then ReverseLinkList(LeftArg);
2061       if not RightArg".ForwardOrder
2062       then ReverseLinkList(RightArg);
2063       NewValTabLink".ForwardOrder := true;
2064       NewValTabLink".dimensions := 1 ( result is a vector );
2065       new(NewDim); NewValTabLink".FirstDimen := NewDim;
2066       NewDim".NextDimen := nil; ResultLength := 0;
2067       if LeftArg".dimensions = 0
2068       then
2069         ResultLength := ResultLength + 1 ( left arg is a scalar )
2070       else
2071         ResultLength := ResultLength + LeftArg".FirstDimen".
2072           dimenlength;
2073       if RightArg".dimensions = 0
2074       then
2075         ResultLength := ResultLength + 1 ( right arg is a scalar )
2076       else
2077         ResultLength := ResultLength + RightArg".FirstDimen".
2078           dimenlength;
2079       NewDim".dimenlength := ResultLength; switch := true;
2080       if ResultLength = 0
2081       then
2082         NewValTabLink".FirstValue := nil
2083         ( result is vector of length 0 )
2084       else
2085         begin ( transfer values to result )
2086           LeftValPtr := LeftArg".FirstValue;
2087           while LeftValPtr <> nil do
2088             begin ( transfer left arg values (if any) )
2089               new(NewValues);
2090               if switch
2091               then

```

```

2092         begin
2093           switch := false;
2094           NewValTabLink".FirstValue := NewValues
2095         end
2096       else NewValPtr".NextValue := NewValues;
2097       NewValues".RealVal := LeftValPtr".RealVal;
2098       NewValPtr := NewValues;
2099       LeftValPtr := LeftValPtr".NextValue
2100     end;
2101     RightValPtr := RightArg".FirstValue;
2102     while RightValPtr <> nil do
2103       begin ( transfer right arg values (if any) )
2104         new(NewValues);
2105         if switch
2106         then
2107           begin
2108             switch := false;
2109             NewValTabLink".FirstValue := NewValues
2110           end
2111         else NewValPtr".NextValue := NewValues;
2112         NewValues".RealVal := RightValPtr".RealVal;
2113         NewValPtr := NewValues;
2114         RightValPtr := RightValPtr".NextValue
2115       end;
2116       NewValues".NextValue := nil
2117     end ( transfer of values )
2118   end
2119 end ( catenate );
2120
2121 procedure IndexOf(LeftArg, RightArg: TypeValTabPtr);
2122 ( dyadic iota operator )
2123
2124 var
2125   MapIndex, icount, TestLength, OneMore: integer;
2126
2127 begin ( indexof )
2128   if LeftArg".dimensions <> 1
2129   then error(29) ( left argument is not a vector )
2130   else
2131     begin
2132       new(NewValTabLink);
2133       OldValTabLink".NextValTabLink := NewValTabLink;
2134       NewValTabLink".NextValTabLink := nil;
2135       NewValTabLink".IntermedResult := true;
2136       if not LeftArg".ForwardOrder
2137       then ReverseLinkList(LeftArg);
2138       NewValTabLink".ForwardOrder := RightArg".ForwardOrder;
2139       NewValTabLink".dimensions := RightArg".dimensions;
2140       if RightArg".dimensions = 0
2141       then
2142         NewValTabLink".FirstDimen := nil
2143         ( right argument is a scalar )
2144       else
2145         begin ( build dimensions of result )
2146           switch := true; DimPtr := RightArg".FirstDimen;
2147           while DimPtr <> nil do
2148             begin
2149               new(NewDim);
2150               if switch
2151               then
2152                 begin
2153                   switch := false;
2154                   NewValTabLink".FirstDimen := NewDim
2155                 end
2156               else NewPtr".NextDimen := NewDim;
2157               NewDim".dimenlength := DimPtr".dimenlength;
2158               NewPtr := NewDim; DimPtr := DimPtr".NextDimen
2159             end;
2160             NewDim".NextDimen := nil
2161           end;
2162           switch := true; RightValPtr := RightArg".FirstValue;
2163           while RightValPtr <> nil do
2164             begin
2165               new(NewValues);
2166               if switch
2167               then
2168                 begin
2169                   switch := false;
2170                   NewValTabLink".FirstValue := NewValues
2171                 end
2172               else NewValPtr".NextValue := NewValues;
2173               icount := 1; LeftValPtr := LeftArg".FirstValue;
2174               TestLength := LeftArg".FirstDimen".dimenlength
2175               ( length of left arg );
2176               OneMore := TestLength + 1
2177               ( length of left arg plus one );
2178               MapIndex := OneMore;
2179               while (icount <= TestLength) and (MapIndex = OneMore) do
2180                 begin
2181                   ( try to match value in right arg with one in left arg )
2182                   if LeftValPtr".RealVal = RightValPtr".RealVal
2183                   then MapIndex := icount ( value match );
2184                   icount := icount + 1;
2185                   LeftValPtr := LeftValPtr".NextValue
2186                 end;
2187               NewValues".RealVal := MapIndex;
2188               NewValPtr := NewValues;
2189               RightValPtr := RightValPtr".NextValue
2190             end
2191           end
2192         ( if no match, index becomes one more than length of left arg )
2193       end
2194     end

```

```

2195     NewValues".NextValue := nil
2196   end
2197 end ( indexof );
198
199
2200 procedure reshape(LeftArg, RightArg: TypeValTabPtr);
2201 ( dyadic rho operator - change dimensions of )
2202
2203 var
2204   ResultLength, elements: integer;
2205   DimPtr: "DimenInfo;
2206   NewPtr: "values;
2207
2208 begin ( reshape )
2209   if LeftArg".dimensions > 1
2210   then error(50) ( left argument not a vector or a scalar )
2211   else
2212     begin
2213       new(NewValTabLink);
2214       OldValTabLink".NextValTabLink := NewValTabLink;
2215       NewValTabLink".NextValTabLink := nil;
2216       NewValTabLink".IntermedResult := true;
2217       if not LeftArg".ForwardOrder
2218       then ReverseLinkList(LeftArg);
2219       if not RightArg".ForwardOrder
2220       then ReverseLinkList(RightArg);
2221       NewValTabLink".ForwardOrder := true;
2222       if LeftArg".FirstDimen = nil
2223       then NewValTabLink".dimensions := 1
2224       else
2225         NewValTabLink".dimensions := LeftArg".FirstDimen".
2226         dimenlength;
2227       ResultLength := 1; LeftValPtr := LeftArg".FirstValue;
2228       switch := true;
2229       while LeftValPtr <> nil do
2230         ( left arg values are dimensions of result )
2231         begin ( build result dimensions )
2232           ResultLength := ResultLength * trunc(LeftValPtr".
2233             RealVal);
2234           new(NewDim);
2235           NewDim".dimenlength := trunc(LeftValPtr".RealVal);
2236           LeftValPtr := LeftValPtr".NextValue;
2237           if switch
2238           then
2239             begin
2240               switch := false;
2241               NewValTabLink".FirstDimen := NewDim
2242             end
2243             else DimPtr".NextDimen := NewDim;
2244               DimPtr := NewDim
2245             end;
2246           NewDim".NextDimen := nil;
2247           RightValPtr := RightArg".FirstValue; elements := 0;
2248           switch := true;
2249           while elements < ResultLength do
2250             begin ( duplicate right arg values into result values )
2251               elements := elements + 1; new(NewValues);
2252               if RightValPtr = nil
2253               ( extend right argument if necessary )
2254               then RightValPtr := RightArg".FirstValue;
2255               NewValues".RealVal := RightValPtr".RealVal;
2256               if switch
2257               then
2258                 begin
2259                   switch := false;
2260                   NewValTabLink".FirstValue := NewValues
2261                 end
2262                 else NewPtr".NextValue := NewValues;
2263                   NewPtr := NewValues;
2264                   RightValPtr := RightValPtr".NextValue
2265                 end;
2266               NewValues".NextValue := nil;
2267             end
2268           end ( reshape );
2269
2270
2271 procedure InnerProduct(LeftArg, RightArg: TypeValTabPtr);
2272
2273 var
2274   Inpro1Code, Inpro2Code, LeftSkip, RightSkip: integer;
2275   icount, jcount, kcount, lcount, mcount: integer;
2276   LastLeftDim, FirstRightDim, CommonLength: integer;
2277   lptr: "values;
2278   hold: real;
2279   SFloat, value: real;
2280
2281 begin ( inner product is matrix multiplication )
2282   DimPtr := LeftArg".FirstDimen;
2283   if LeftArg".FirstDimen <> nil then
2284     while DimPtr".NextDimen <> nil do
2285       DimPtr := DimPtr".NextDimen
2286       ( get last dimen of left arg(if any) );
2287   if (DimPtr <> nil) and (RightArg".FirstDimen <> nil)
2288   then
2289     if DimPtr".dimenlength <> RightArg".FirstDimen".dimenlength
2290     then
2291       error(52)
2292       ( last dim of left arg not = to first dim of right arg )
2293     else
2294       begin
2295         Inpro1Code := code div 100 ( separate operators );
2296         Inpro2Code := code - 100 * Inpro1Code;
2297         new(NewValTabLink);
2298         OldValTabLink".NextValTabLink := NewValTabLink;
2299         NewValTabLink".NextValTabLink := nil;
2300         NewValTabLink".IntermedResult := true;
2301         if not LeftArg".ForwardOrder
2302         then ReverseLinkList(LeftArg);
2303         if not RightArg".ForwardOrder
2304         then ReverseLinkList(RightArg);
2305         NewValTabLink".ForwardOrder := true;
2306         NewValTabLink".dimensions := LeftArg".dimensions +
2307         RightArg".dimensions - 2;
2308         if NewValTabLink".dimensions < 0
2309         then NewValTabLink".dimensions := 0;
2310         switch := true; LastLeftDim := 0;
2311         if LeftArg".FirstDimen <> nil
2312         then
2313           begin ( copy all but last of left arg dims into result )
2314             LeftSkip := 1; DimPtr := LeftArg".FirstDimen;
2315             while DimPtr".NextDimen <> nil do
2316               begin ( copy left arg dimensions )
2317                 new(NewDim);
2318                 NewDim".dimenlength := DimPtr".dimenlength;
2319                 LeftSkip := LeftSkip + DimPtr".dimenlength;
2320                 if switch
2321                 then
2322                   begin
2323                     switch := false;
2324                     NewValTabLink".FirstDimen := NewDim
2325                   end
2326                   else NewPtr".NextDimen := NewDim;
2327                     NewPtr := NewDim; DimPtr := DimPtr".NextDimen
2328                   end;
2329                 LastLeftDim := DimPtr".dimenlength
2330               end;
2331             if RightArg".FirstDimen <> nil
2332             then
2333               begin
2334                 ( copy all but first of right arg dims into result )
2335                 RightSkip := 1;
2336                 DimPtr := RightArg".FirstDimen".NextDimen;
2337                 while DimPtr <> nil do
2338                   begin ( copy right arg dimensions )
2339                     new(NewDim);
2340                     NewDim".dimenlength := DimPtr".dimenlength;
2341                     RightSkip := RightSkip + DimPtr".dimenlength;
2342                     if switch
2343                     then
2344                       begin
2345                         switch := false;
2346                         NewValTabLink".FirstDimen := NewDim
2347                       end
2348                       else NewPtr".NextDimen := NewDim;
2349                         NewPtr := NewDim; DimPtr := DimPtr".NextDimen
2350                       end
2351                     end;
2352                   if switch then NewValTabLink".FirstDimen := nil
2353                   else NewDim".NextDimen := nil;
2354                   if LeftArg".FirstValue = nil then LeftSkip := 0;
2355                   if RightArg".FirstValue = nil then RightSkip := 0;
2356                   switch := true;
2357                   if RightArg".FirstDimen <> nil
2358                   then FirstRightDim := RightArg".FirstDimen".dimenlength
2359                   else FirstRightDim := 0;
2360                   if FirstRightDim > LastLeftDim
2361                   then CommonLength := FirstRightDim
2362                   else CommonLength := LastLeftDim;
2363                   icount := 0; LeftValPtr := LeftArg".FirstValue;
2364                   while icount < LeftSkip do
2365                     begin ( loop for each row in left arg )
2366                       lptr := LeftValPtr ( hold start of row position );
2367                       jcount := 0;
2368                       while jcount < RightSkip do
2369                         begin ( loop for each column in right arg )
2370                           LeftValPtr := lptr;
2371                           RightValPtr := RightArg".FirstValue;
2372                           lcount := 0;
2373                           while lcount < jcount do
2374                             begin ( skip to starting value in right arg )
2375                               RightValPtr := RightValPtr".NextValue;
2376                               if RightValPtr = nil then
2377                                 RightValPtr := RightArg".FirstValue
2378                                 ( extend arg );
2379                               lcount := lcount + 1
2380                             end;
2381                             kcount := 0;
2382                             while kcount < CommonLength do
2383                               begin ( loop for each element in row/column )
2384                                 SFloat := RightValPtr".RealVal;
2385                                 DyadComp(SFloat, LeftValPtr".RealVal,
2386                                   Inpro2Code);
2387                                 value := SFloat;
2388                                 if kcount = 0
2389                                 then
2390                                   ( set identity value for first time through )
2391                                   case Inpro1Code of
2392                                     52, 53, 78: SFloat := 0.0;
2393                                     54, 55, 56, 77: SFloat := 1.0;
2394                                     71, 72, 73, 74, 75, 76: ( null case )
2395                                     end ( case )
2396                                   else SFloat := hold;
2397                                 DyadComp(SFloat, value, Inpro1Code);
2398                                 hold := SFloat ( save summer result );

```

```

2399 LeftValPtr := LeftValPtr.NextValue;
2400 if LeftValPtr = nil then
2401   LeftValPtr := LeftArg.FirstValue
2402   ( extend arg );
2403 mcount := 0;
2404 while mcount < RightSkip do
2405   begin ( skip to next value in right arg )
2406     mcount := mcount + 1;
2407     RightValPtr := RightValPtr.NextValue;
2408     if RightValPtr = nil
2409     then RightValPtr := RightArg.FirstValue;
2410     end;
2411     kcount := kcount + 1
2412   end;
2413 new(NewValues); NewValues.RealVal := SFloat;
2414 if switch
2415 then
2416   begin
2417     switch := false;
2418     NewValTabLink.FirstValue := NewValues
2419   end
2420 else NewValPtr.NextValue := NewValues;
2421   NewValPtr := NewValues; jcount := jcount + 1;
2422   end;
2423   icount := icount + 1
2424   end;
2425   if switch then NewValTabLink.FirstValue := nil
2426   else NewValues.NextValue := nil
2427   end
2428 end ( Innerproduct );
2429
2430 procedure OuterProduce(LeftArg, RightArg: TypeValTabPtr);
2431 var
2432   OutProCode: integer;
2433   SFloat: real;
2434 begin
2435   OutProCode := code div 10; new(NewValTabLink);
2436   OldValTabLink.NextValTabLink := NewValTabLink;
2437   NewValTabLink.NextValTabLink := nil;
2438   NewValTabLink.IntermedResult := true;
2439   if not LeftArg.ForwardOrder
2440   then ReverseLinkList(LeftArg);
2441   if not RightArg.ForwardOrder
2442   then ReverseLinkList(RightArg);
2443   NewValTabLink.ForwardOrder := true;
2444   NewValTabLink.dimensions := LeftArg.dimensions + RightArg.
2445   dimensions;
2446   switch := true; DimPtr := LeftArg.FirstDimen;
2447   while DimPtr <> nil do
2448     begin ( copy left arg dimensions to result )
2449       new(NewDim); NewDim.dimenlength := DimPtr.dimenlength;
2450       if switch
2451       then
2452         begin
2453           switch := false; NewValTabLink.FirstDimen := NewDim
2454         end
2455       else NewPtr.NextDimen := NewDim;
2456           NewPtr := NewDim; DimPtr := DimPtr.NextDimen
2457         end;
2458       DimPtr := RightArg.FirstDimen;
2459       while DimPtr <> nil do
2460         begin ( copy dimensions of right arg to result )
2461           new(NewDim); NewDim.dimenlength := DimPtr.dimenlength;
2462           if switch
2463           then
2464             begin
2465               switch := false; NewValTabLink.FirstDimen := NewDim
2466             end
2467           else NewPtr.NextDimen := NewDim;
2468               NewPtr := NewDim; DimPtr := DimPtr.NextDimen
2469             end;
2470           if switch then NewValTabLink.FirstDimen := nil
2471           else NewDim.NextDimen := nil;
2472           switch := true; LeftValPtr := LeftArg.FirstValue;
2473           while LeftValPtr <> nil do
2474             begin
2475               RightValPtr := RightArg.FirstValue;
2476               while RightValPtr <> nil do
2477                 begin
2478                   SFloat := RightValPtr.RealVal;
2479                   DyadComp(SFloat, LeftValPtr.RealVal, OutProCode);
2480                   new(NewValues);
2481                   if switch
2482                   then
2483                     begin
2484                       switch := false;
2485                       NewValTabLink.FirstValue := NewValues
2486                     end
2487                   else NewValPtr.NextValue := NewValues;
2488                       NewValues.RealVal := SFloat; NewValPtr := NewValues;
2489                       RightValPtr := RightValPtr.NextValue
2490                     end;
2491                   LeftValPtr := LeftValPtr.NextValue
2492                 end;
2493               end;
2494               if switch then NewValTabLink.FirstValue := nil
2495               else NewValues.NextValue := nil
2496             end ( outerproduct );
2497           end

```

```

2500 procedure dyadic(LeftArg, RightArg: TypeValTabPtr);
2501 ( operators with codes of 52 and higher )
2502 var
2503   compatible: Boolean;
2504   arg: TypeValTabPtr;
2505   SFloat: real;
2506 begin
2507   if code > 1000 then InnerProduct(LeftArg, RightArg)
2508   else
2509     if code > 100 then OuterProduce(LeftArg, RightArg)
2510     else
2511       if code > 80
2512       then
2513         case code of
2514           87: IndexOf(LeftArg, RightArg);
2515           88: reshape(LeftArg, RightArg);
2516           89: catenate(LeftArg, RightArg)
2517         end ( case )
2518       else
2519         begin ( simple dyadics )
2520           compatible := true;
2521           if (LeftArg.dimensions >= 1) and (RightArg.
2522             dimensions >= 1)
2523           then
2524             if LeftArg.dimensions <> RightArg.dimensions
2525             then
2526               compatible := false
2527               ( different ranks/neither scalar )
2528             else
2529               begin ( ranks match - check lengths )
2530                 LeftDimPtr := LeftArg.FirstDimen;
2531                 RighthDimPtr := RightArg.FirstDimen;
2532                 while LeftDimPtr <> nil do
2533                   begin
2534                     if LeftDimPtr.dimenlength <> RighthDimPtr.
2535                     dimenlength
2536                     then
2537                       compatible := false ( different length(s) );
2538                       LeftDimPtr := LeftDimPtr.NextDimen;
2539                       RighthDimPtr := RighthDimPtr.NextDimen
2540                     end
2541                   end;
2542                 if compatible
2543                 ( arguments suitable for dyadic operation )
2544                 then
2545                   begin ( build dimensions of result )
2546                     if RightArg.dimensions > LeftArg.dimensions
2547                     then arg := RightArg
2548                     else
2549                       arg := LeftArg ( result has shape of larger arg );
2550                     new(NewValTabLink);
2551                     OldValTabLink.NextValTabLink := NewValTabLink;
2552                     NewValTabLink.NextValTabLink := nil;
2553                     NewValTabLink.IntermedResult := true;
2554                     if LeftArg.ForwardOrder <> RightArg.ForwardOrder
2555                     then ReverseLinkList(LeftArg);
2556                     NewValTabLink.ForwardOrder := arg.ForwardOrder;
2557                     NewValTabLink.dimensions := arg.dimensions;
2558                     switch := true; DimPtr := arg.FirstDimen;
2559                     while DimPtr <> nil do
2560                       begin ( copy dimensions to result )
2561                         new(NewDim);
2562                         NewDim.dimenlength := DimPtr.dimenlength;
2563                         if switch
2564                         then
2565                           begin
2566                             switch := false;
2567                             NewValTabLink.FirstDimen := NewDim
2568                           end
2569                         else NewPtr.NextDimen := NewDim;
2570                             NewPtr := NewDim;
2571                             DimPtr := DimPtr.NextDimen
2572                           end;
2573                         if switch
2574                         then
2575                           NewValTabLink.FirstDimen := nil
2576                           ( result is a scal )
2577                         else NewDim.NextDimen := nil;
2578                             switch := true;
2579                             RightValPtr := RightArg.FirstValue;
2580                             LeftValPtr := LeftArg.FirstValue;
2581                             ValPtr := arg.FirstValue;
2582                             while ValPtr <> nil do
2583                               begin ( perform operation )
2584                                 new(NewValues);
2585                                 SFloat := RightValPtr.RealVal;
2586                                 DyadComp(SFloat, LeftValPtr.RealVal, code);
2587                                 NewValues.RealVal := SFloat;
2588                                 if switch
2589                                 then
2590                                   begin
2591                                     switch := false;
2592                                     NewValTabLink.FirstValue := NewValues
2593                                   end
2594                                 else NewValPtr.NextValue := NewValues;
2595                                     NewValPtr := NewValues;
2596                                     ValPtr := ValPtr.NextValue;
2597                                     LeftValPtr := LeftValPtr.NextValue;
2598                                   end
2599                               end

```

```

2601 RightValPtr := RightValPtr.NextValue;
2602 if LeftValPtr = nil then
2603   LeftValPtr := LeftArg.FirstValue
2604   ( extend arg );
2605   if RightValPtr = nil then
2606     RightValPtr := RightArg.FirstValue
2607     ( extend );
2608   end;
2609   if switch
2610   then
2611     NewValTabLink.FirstValue := nil
2612     ( vector of len 0 )
2613   else NewValues.NextValue := nil
2614   end
2615   else
2616     error(55)
2617     ( arguments incompatible for dyadic operation )
2618   end
2619 end ( dyadic );
2620
2621 procedure FunCall(var ValidFunk: Boolean);
2622 var
2623   ValidPm: Boolean;
2624 begin ( funcall )
2625   ValidFunk := false;
2626   if FunctCall
2627   then
2628     begin
2629       if TokenTabPtr.noun <> StatEnd
2630       then
2631         begin
2632           SubrTabPtr.TokenCallingSubr := TokenTabPtr;
2633           primary(ValidPm); if not ValidPm then error(17);
2634           ( 'leftarg of dyadic func call not a primary' )
2635         end;
2636         CallSubr; ValidFunk := true;
2637       end;
2638     end ( funcall );
2639
2640 begin ( expression )
2641   primary(ValidPri);
2642   if not ValidPri
2643   then
2644     begin
2645       if TokenTabPtr.noun = StatEnd
2646       then begin ValidExp := true; assign1 := true end
2647       else ValidExp := false
2648       end
2649     else
2650     begin
2651       DoneExp := false;
2652       while not DoneExp do
2653       begin
2654         FunCall(ValidFunk);
2655         if ValidFunk
2656         then begin expression(ValidExp); DoneExp := true end
2657         else
2658         begin
2659           assignment(ValidAssn);
2660           if ValidAssn and (TokenTabPtr.noun = StatEnd)
2661           then begin DoneExp := true; ValidExp := true; end;
2662           if not ValidAssn
2663           then
2664             if mop
2665             then
2666               begin
2667                 monadic(OperTabPtr.OperPtr, hold);
2668                 OperTabPtr.OperPtr := NewValTabLink
2669               end
2670             else
2671             if not dop
2672             then begin ValidExp := true; DoneExp := true end
2673             else
2674             begin
2675               primary(ValidPri);
2676               if not ValidPri
2677               then
2678                 error(13)
2679                 ( dyad oper not preceded by a pri )
2680             else
2681             begin
2682               dyadic(OperTabPtr.OperPtr, OperTabPtr.
2683                 LastOper.OperPtr);
2684               AuxOperTabPtr := OperTabPtr;
2685               OperTabPtr := OperTabPtr.LastOper;
2686               PtrLastOper := OperTabPtr;
2687               dispose(AuxOperTabPtr);
2688               OperTabPtr.OperPtr := NewValTabLink;
2689             end;
2690           end;
2691         end;
2692       end;
2693     end;
2694   end;
2695 end;
2696 end ( expression );

```

```

2701 begin ( parser )
2702 assign := false; assign1 := false; DoneParse := false;
2703 repeat
2704   expression(ValidExp) ( checks for valid expression );
2705   if not ValidExp then error(10) ( 'invalid expression' )
2706   else
2707     if SpecSymbol(XRightArrow)
2708     then
2709       if not ((OperTabPtr.OperPtr.FirstValue = nil) and (
2710         OperTabPtr.OperPtr.dimensions > 0))
2711       then ( branch )
2712         ( result of expression is at opertabptr )
2713         if OperTabPtr.OperPtr.FirstValue.RealVal - 1.0 * trunc
2714         (OperTabPtr.OperPtr.FirstValue.RealVal) <> 0.0
2715         then error(12) ( stmt.num.to branch to not an integer )
2716         else
2717           if SubrTabPtr = nil
2718           then
2719             begin ( function mode )
2720               TokenTabPtr := hold; DoneParse := true
2721             end
2722           else
2723             if trunc(OperTabPtr.OperPtr.FirstValue.RealVal) in
2724             [1 .. (SubrTabPtr.CalledSubr.NumOfStatements)]
2725             then
2726               begin
2727                 VFuncHold := SubrTabPtr.CalledSubr.FirstStatement;
2728                 for cnt := 1 to trunc(OperTabPtr.OperPtr.
2729                   FirstValue.RealVal) do
2730                   begin
2731                     VFuncPtr := VFuncHold;
2732                     TokenTabPtr := VFuncPtr.NextStmnt;
2733                     VFuncHold := VFuncPtr.NextVFuncPtr
2734                   end;
2735                     AuxOperTabPtr := OperTabPtr;
2736                     OperTabPtr := OperTabPtr.LastOper;
2737                     dispose(AuxOperTabPtr); PtrLastOper := OperTabPtr;
2738                     TokenTabPtr := VFuncPtr.NextStant
2739                   end
2740                 else ReturnToCallingSubr
2741             else ( successor )
2742             else ( successor )
2743             begin
2744               if not assign1 then OutPutVal; assign1 := false;
2745               if SubrTabPtr = nil
2746               then
2747                 begin ( interpretive )
2748                   hold := TokenTabPtr;
2749                   TokenTabPtr := TokenTabPtr.NextToken;
2750                   DoneParse := true
2751                 end
2752             else ( function )
2753             begin
2754               VFuncPtr := VFuncPtr.NextVFuncPtr;
2755               DoneSuccessor := false;
2756               repeat
2757                 if VFuncPtr <> nil
2758                 then
2759                   begin
2760                     TokenTabPtr := VFuncPtr.NextStmnt;
2761                     DoneSuccessor := true
2762                   end
2763                 else
2764                 begin
2765                   ReturnToCallingSubr;
2766                   if TokenTabPtr.noun = StatEnd
2767                   then DoneSuccessor := true;
2768                 end;
2769               until DoneSuccessor;
2770             end;
2771           end
2772         until DoneParse;
2773         release ( release memory );
2774       end ( parser );
2775
2776 begin ( scanner )
2777 initializeCharacterSet; ReadInErrorMsgs;
2778 InitParser ( initialize tables etc. ); FillUpTables;
2779 FunctionMode := false; FirstFunction := true;
2780 OldValTabLink := nil; OldFuncTabPtr := nil; OldVarTabPtr := nil;
2781 OldTokenPtr := nil; NewTokenPtr := nil; NewFuncTabPtr := nil;
2782 NewVFuncPtr := nil; HoldTokenPtr := nil; TokenError := false;
2783 NewValTabLink := nil; NewVarTabPtr := nil; GetAPLstatement;
2784 while (APLstatement[1] <> character[ForwardSlash]) or (APLstatement[2]
2785   <> character[Lasterisk]) do ( ' ends program ' )
2786   begin
2787     SkipSpaces; TokenSwitch := true;
2788     while (position <= LineLength) and (not TokenError) and (not
2789       LineTooLong) do
2790       begin ( scanning )
2791         if APLstatement[position] = character[del]
2792         ( function delimiter )
2793         then ( del encountered )
2794         ( if functionMode
2795         then
2796           begin ( end of current function )
2797             if NewFuncTabPtr <> nil
2798             then NewFuncTabPtr.NumOfStatements := FuncStatements;
2799             if FuncStatements > 0

```


