$10.00

Pascal & MODULA2

Computer Graphic Design by Catherine Del Tito

Number 27
November 83

ISSN 0739-1900

SUBVERSIVE SOFTWARE

So cheap and useful it's...dangerous!

A radical idea in software development.
Useful Pascal programs, with:

- Complete annotated *source listings;*
- Disk(s) with *source code and compiled code;*
- User manual;
- Complete programmer documentation describing data structures and algorithms; and giving suggestions for modification.

Modify them, include them in your own systems, or simply use them.

A growing library of software tools you'll find hard to resist.

SUBVERSIVE SOFTWARE

TPL

The Text Processing Language. A text-file runoff program consisting of a set of text-processing primitive commands from which more complex commands (macros) can be built (as in Logo). Features include:

- Complete customization of text processing through macro definition and expansion, looping structures, and conditional statements;
- Adapts to any printer;
- Pagination;
- Text justification and centering;
- Indexing and tables of contents;
- Superscripts and subscripts;
- Bolding and underlining;
- Multiple headers and footers;
- End notes and footnotes;
- Widow and orphan suppression;
- Floating tables and 'keeps.'

$50

SUBVERSIVE SOFTWARE

DBX

Blocked Keyed Data Access Module. Maintains disk files of keyed data. Can be used for bibliographies, glossaries, multi-key data base construction, and many other applications.

- Variable-length keys;
- Variable-length data;
- Sequential access and rapid keyed access;
- Single disk access per operation (store, find, delete) in most cases;
- Multiple files;
- Dynamic memory allocation for RAM-resident index and current "page" of entries;
- Includes demonstration program and testbed program.

$50

SUBVERSIVE SOFTWARE

PDMS

The Pascal Data Management System. A user-oriented data management system in which numeric and alphanumeric data are stored in tables with named columns and numbered rows. Currently being used for dozens of different kinds of business and scientific applications, from inventory management to laboratory data analysis. Includes over 20 Pascal programs; more than 10,000 lines of code. Main features include:

- Maximum of 32,767 rows per file;
- Maximum of 400 characters per row, and 40 columns per table;
- Full-screen editing of rows and columns, with scrolling, windowing, global search/replace, and other editing features;
- Sorting, copying, merging, and reducing routines;
- Mailing label program;
- Reporting program generates reports with control breaks, totals and subtotals, and selects rows by field value; many other reporting features;
- Cross-tabulation, correlations, and multiple regression;
- Video-display-handling module;
- Disk-file-handling module.

Many other features. UCSD formats only.

$250

SUBVERSIVE SOFTWARE

ZED

Full-screen text editor; designed to be used either with TPL or by itself.

- Full cursor control;
- Insert mode with word wrap;
- 'Paint' mode;
- Single-keystroke or dual-keystroke commands;
- Command synonyms;
- Global search and replace;
- Block move, block copy, and block delete.

$50

BVERSIVE FTWARE

SCINTILLA

A log logit curve fitting program for radio-immunologic data; must be used with PDMS (described above).

- Multiple protocol files;
- Quality control files;
- Four-parameter non-linear curve fit.

UCSD formats only. **$250**

SUBVERSIVE SOFTWARE

CHROME

Chromatography data analysis program:

- Graphic display of analog data;
- Panning and zooming;
- Automatic peak-finding and baseline calculation;
- Full interactive peak editing;
- Computation of peak areas;
- Strip charts on C. Itoh and EPSON printers.

$100

SUBVERSIVE SOFTWARE

PLANE

Planimetry program:

- Bit-pad entry of cross sections;
- Real-time turtlegraphics display;
- Calculation of areas;
- Saves calculations to text file.

$100

VERSIVE TWARE

MINT

A terminal emulation program for communication between computers of any size.

- User-configurable uploading and downloading of files;
- X-ON/X-OFF and EOB/ACK protocols;
- Interrupt-driven serial input (for Prometheus Versacard in Apple II);
- Printer-logging.

$50

For more information, call 919-942-1411. To order, use form below or call our toll-free number: 1-800-X-PASCAL.

(In N.C. use 1-800-642-0949)

Check appropriate boxes:

FORMAT	PRODUCT	PRICE
☐ 8″ UCSD SSSD	☐ DBX	$ 50
☐ 5¼″ Apple Pascal	☐ PDMS	$250
☐ 5¼″ UCSD IBM PC 320k	☐ TPL	$ 50
☐ 8″ CP/M SSSD	☐ ZED	$ 50
☐ 5¼″ IBM MS-DOS	☐ MINT	$ 50
☐ 5¼″ CP/M Osborne	☐ SCINTILLA	$250
	☐ CHROME	$100
	☐ PLANE	$100

Name ____________________

Address ____________________

☐ MasterCard ☐ VISA ☐ Check ☐ C.O.D.

(Please include card # and expiration date)

SUBVERSIVE SOFTWARE A division of Pascal & Associates,
135 East Rosemary St., Chapel Hill, NC 27514

pple and Apple Pascal are trademarks of the APPLE Computer Corp. IBM and IBM PC are trademarks of International Business lachines. UCSD Pascal is a trademark of the Regents of the University of California. Osborne is a trademark of Osborne Comuter. EPSON is a trademark of EPSON America, Inc. C. Itoh is a trademark of C. Itoh Electronics.

Pascal & MODULA-2

Formerly *Pascal News*

Serving Pascal Users Group and the Modula-2 Users Group

November 1983 **Number 27**

3 — **EDITORIAL**

5 — **OPEN FORUM**

8 — **Two Pascal Devices**
by Harley Flanders, Florida Atlantic University

9 — **Zuse User's Manual**
By Arthur Pyster, University of California

33 — **ANNOUNCEMENTS**

37 — **MODULA-2 ANNOUNCEMENTS**

41 — **ORDER FORMS**

About the cover:
The cover computer graphics were created by computer artist Catherine Del Tito. The program was written for an Apple IIe and a Vectrix computer system.

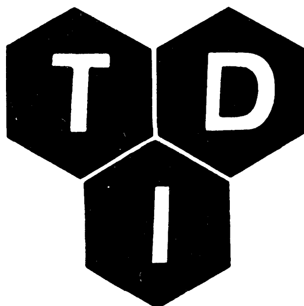
UCSD p-SYSTEM™
with UCSD PASCAL™
FOR THE
VICTOR 9000™

Standard Features

- System Foundation:
  - Operating System - Single key commands
  - Filer - For file management
  - Screen Editor - Powerful text editor
  - Utility Library - Fast development tools
- **UCSD PASCAL Compiler** - Only requires 128K
- Utilities for the Victor:
  - Keyboard Editor - Define your own keyboard
  - Character Editor - Design character sets
  - Config - Define machine parameters
  - Diskutil - Disk copy/format
  - Remote - File porting
- Native Code Generator - For faster execution
- Turtlegraphics - 800 × 400 pixel access
- RAMDISK Capability - For RAM above 128K
- Assembler - 8086 with macros
- Documentation:
  - Owner's Manual
  - User Manual/Supplement
  - Architecture Guide
  - Installation Guide Excerpts
  - DEMODOC diskette - On line tutorial for p-System overview

Optional Features

- Hard Disk Support Software - For internal hard disk
- FORTRAN 77 Compiler - Only requires 128K
- BASIC Compiler
- 8087 Support Software
- Advanced Systems Editor
- Xenofile™ - Convert CP/M files
- Applications Software (call for information)



TDI SYSTEMS, INC.
620 Hungerford Drive
Suite 33
Rockville, Maryland 20850
(301) 340-8700

TDI LIMITED
29 Alma Vale Road
Bristol, U.K. BS8 2HL
0272 742 796

UCSD p-SYSTEM and UCSD Pascal are trademarks of the Regents of the University of California
Universal Operating System and Xenofile are trademarks of SofTech Microsystems, Inc.
Victor 9000 is a trademark of Victor Technologies, Inc.

Editor's Notes

Announcing a $10,000 Contest for the Best Article or Application Voted by the Members

Dear Member:

By now those of you in the United States have received a questionnaire and announcement concerning *"Pascal News."* The reader survey is a very important element in our ability to contain the cost of this newsletter. When we solicit advertisers, they want to know about the members: Who we are, What we do, How much we spend. I realize these are very probing questions and you may feel they are too personal. To separate your name from your information, the subscription card is a self mailer and the return envelope for the survey will assure your anonymity. Thank you in advance for your help.

Let me explain the announcement of our name change from *"Pascal News"* to *"Pascal & Modula2."* Pascal has encouraged and endorsed rational programming. The language aids in the segmentation of a job into small parts through procedures and functions. I have enjoyed learning and using this language, but Pascal does have limitations.

Many limitations are part of the design of a teaching language. Pascal is a very nice base, a core, to which extensions have been added by every implementor. We have accepted this and published reports of large extension packages, UCSD Pascal, Concurrant Pascal and Path Pascal.

These extensions were added to a language created for teaching programming principles. This design goal allows a general purpose language, but not an all purpose language. Nicklaus Wirth recognized these limitations and created a language that would contain Pascal's good features and satisfy the goal of an all purpose language.

Modula-2, created in 1977, is that language. In this issue, you will read Modula-2 product announcements. In these announcements, and in other articles, claims are made that Pascal is finished, that Modula-2 should replace Pascal in all cases. Maybe. I believe Pascal can and should remain in the position for which it was designed. The premier teaching language is Pascal. The easy transition from Pascal to Modula-2 makes Modula-2 an excellent second year language. Restrictions are necessary in the introduction to programming and Modula-2's flexibility does not focus a student to basic principles.

Of course I may be "all wet," but I believe Pascal should remain.

Pascal's teaching tool strength, Modula-2's all purpose ability, and the relatively painless transition between them make Pascal and Modula-2 proper subjects for the Pascal Users Group. Pascal News should reflect this wider interest in content and name. The new name, *"Pascal & Modula2,"* keeps Pascal as the first name and can be found in the same place in periodical indices as *"Pascal News."* This should make the libraries happy.

The new logo places Pascal within Modula2, a reference to Modla-2's ability for operating system programming, above Pascal, and its ability for machine specific programming, below Pascal. I hope you welcome the change and contribute to the discussion and promotion.

This brings me to the contest for $10,000. It really is a promotion. There are many ways to promote this newsletter and I have tried a few. One way is to keep the members we have now. I have promised four issues per year and this is the fourth of 1983. A renewal notice was sent out in January and I thank you for your subscription. In October the reader survey and subscription card were mailed and I hope you renew promptly. I placed a small ad in *"PC"* magazine. It is in their Blue Book section and will run from September 1983 through March 1984 and should attract new members. Announcement of our name change and subscription information was sent to fifteen magazines. Unfortunately there are many Pascalers who do not know of the Pascal Users Group.

I am now very familiar with the costs of this newsletter and can say that income closely matches costs. I also know that if membership exceeds 5,000, we will have a surplus. What would we do with this money? Well, I propose that it be used for a promotion, and I cannot think of a better promotion than to vote for the best article or application published in *"Pascal & Modula2."*

I do not know whether the prize will be winner take all or first, second and third place division of the money. Your letters will help me form the rules.

The first rule is we must have 5000 members. Fewer members and we cannot afford this contest.

Now, you may recognize a little circularity here. The promotion attracts members and increased membership allows the promotion. Because of my other responsibilities (wife, home, job and country) I need your help to announce this contest in all quarters. If all 4000 members will send one letter (i.e. to friends, to users groups, to fellow students, to magazines) to announce this contest and one new member joins per letter, well, I think you get the idea.

Rule number two—all contestants must be members.

One more idea. If this contest generates enough material and money the newsletter will be published more often. Please send your ideas.

Editor's Notes

Pascal at Work

A typesetting program called TeX, created by Donald Knuth, is available for the cost of distribution. 15,000 lines of Pascal code puts TeX in the nontrivial category.

Basic information is available from two sources. The book *"TeX and METAFONT"* is available for $12. Digital Press, 12A Esquire Road, North Billerica, MA 01862. A new book, *"TeX Book,"* should be in print by the end of 1983 from Addison-Wessley.

Continuing information is provided by the TeX Users Group (TUG). Membership is $30. TeX Users Group, c/o American Mathematical Society, PO Box 1571 Annex Station, Providence, Rhode Island 02901, USA.

"Small Talk," a language from the Xerox Palo Alto Research Center was revealed in the August 1981 issue of *"Byte"* magazine. The Small Talk virtual machine, a software interface to the real machine, was given to four companies. They agreed to debug the virtual machine and share information.

I found it interesting that Tektronics implemented the virtual machine in Pascal. I understand that Tektronics internal programming uses Pascal or Modula-2. The papers regarding the Small Talk research are assembled in the book *"Small Talk-80 Bits of History, Words of Advice."* This book and *"Small Talk-80 The Language and its Implementation"* are available from Addison-Wessley.

Andy Michel informs me of a version of "C," the Bell Labs language implemented in Pascal. Very interesting.

This Issue

A large part of this issue is devoted to a compiler creator. With a language specification this program will write the compiler. I have been told this program is very valuable. Using it, a contract for a compiler was satisfied and a handsome profit gained. (Sounds good to me.)

Robert Gustafson in a letter complains about the typesetting of this newsletter. Expense, time and typographical errors are his concern. When printing more than 2500 copies, typesetting reduces overall expenses. Typesetting consumes 3 weeks' time, not an unreasonable amount for a quarterly.

Typographical errors are a problem and programs are photographed from originals to avoid them. A better way to assure correct information and dark, clean type is to capture the author's original keystrikes. Floppy disks and mag tape will do the job, but I would have to convert the many magnetic formats to one the typesetter could use.

Computer networks may be a better solution. Networks force all submissions to a common format. From this, the typesetter will make only one conversion to his format. If there is a consensus, I will establish a bulletin board and file system on the most popular network.

Please send your ideas. This is a one-man operation and I appreciate and need your help.

I hope you enjoy this issue.

Charlie

2903 Huntington Road
Cleveland, Ohio 44120

Publisher and Editor:
Charles Gaffney

General Consultants:
Studio Graphics Advertising

Production Manager:
Spence Coghlan

National Sales Representative:
John Bachmann

The *Pascal & Modula2* is published for the Pascal Users Group and Modula-2 Users Group, 2903 Huntington Rd., Cleveland, Ohio 44120. *Pascal & Modula2* is a direct benefit of membership.

Membership dues are $25.00 U.S. regular, other forms of membership please inquire. Inquiries regarding membership should be sent to the above address. Magazine correspondence and advertising should be sent to the editor at the aforementioned address. Advertising rates are also available from the above address.

Open Forum

August 22, 1983

Dear Charlie:

I have been a subscriber to *Pascal News* for a long time. According to the date on my mailing label, it looks like I will continue to be a subscriber well into the distant (85) future.

I agree with many of the notes published in *PN #26* concerning your contributions to the well being of *Pascal News*, particularly the general management and timely publication.

However, I believe that *PN* is doomed to die because of one change you have made. Previously, the magazine was printed using the original letters submitted by *PN* correspondents. This was an inexpensive method of getting the information out to the readers. Your current system of rekeyboarding all of the correspondence can only be costing you dollars without increasing the utility of the information. Also, as people decide they can do without *PN* for $25/year, your circulation will decrease. One of the previous attractions of *PN* was that since it was so cheap, it reached **everyone**. A reduced circulation will cause contributors to look for other distribution media. I realize you are saving money on postage, but since *PN* is sent out bulk rate, the savings are only a small fraction of the cost of typesetting. Since these typesetting costs are the same whether you send out 1 copy or 3000 copies, I predict that you will not be able to compete with other publications in the field and *PN* will die.

If you return to direct printing of correspondent copy you will be in good company. The DECUS special interest group newsletters (RSX SIG for example) are printed from yechy LA36 printer copy. I read them cover to cover as soon as they arrive. There are also a lot of expensive stock market newsletters which are printed from typewriter copy. These sell for big bucks because the buyers are interested in TIMELY information, not typography and art work suitable only for decorating a coffee table.

By concentrating on quick publication, I believe you will find advertisers willing to pay for space in *PN*. Also, when it comes to selling advertising space, 3000 subscribers are much better than 300! Another important criterion for an advertiser is knowing that what is sent to you will appear unchanged in the publication.

I have noticed that there are a number of typographical errors in the typeset issues of *PN*. These are inevitable in a keyed-once, proof-read-once publication (I know the statistics, one of our programming systems typesets approximately 40% of the municipal bond issues in the U.S. Errors here are a no-no. As a consequence, we have done quite a bit with automatic proofreading and word processor telecommunications). If you use advertiser copy directly, there can be no problems later with omissions of important parts of the copy or inadvertent changs to prices, etc.

You might consider including a questionnaire in the next issue of *PN*. I'm sure that the majority of your current subscribers would be much more enthusiastic about paying $9/year for the latest information from your correspondents and advertisers printed "as is" than $25/year for the present "remassaged" format. If you choose an even lower price, advertisers will pay more for the resulting increased circulation. Simple economics!

Sincerely,

Robert D. Gustafson
President
Simulation Specialists Inc.
609 West Stratford Place
Chicago, IL 60657

June 29, 1983

Dear Editor:

I enclose a short article, *Two Pascal Devices*, for publication in *Pascal News*. I shall appreciate your acknowledgement and, if you use the article, a copy of the issue in which it is printed.

Sincerely yours,

Harley Flanders
Professor of Mathematics
Florida Atlantic University
Boca Raton, FL 33431

P.S. May I humbly suggest that you do not print the output of line printers or dot matrix printers. It is just too hard to read.

July 5, 1983

Dear Mr. News:

This is to request address correction from that found on the label:
Jeff Davis [81]
135 Turtle Creek
#1 Roper Mt. Rd.
Greenville, SC 29603
to the following:
Jeff Davis
6549 Quiet Hours Apt. #201
Columbia, MD 21045

I am entering this using an editor found in "Software Tools in Pascal," one of the best books on toolbuilding I've ever read, and printing out using a copy of "Prose" from an early copy of *Pascal News* recompiled on an Apple ///! As you see, my interest is in learning by building tools in Pascal.

By the way, there is a local bulletin Board system (my next tool may be a spelling checker!) called Magus which is actually an operating system written in Pascal by Craig Vaughn that is worthy of note. I'll suggest that he submit an article describing it and see what his reaction is.

As a last comment, I had been out of the country for a few years and only recently resubscribed. How's Modula-2 doing in the states? I've ordered their documentation but version for my computer (Apple ///) somewhat tardy.

Thanks and it's great to be back in touch with Pascal reality again!

Sincerely,
Jeff Davis

May 31, 1983

Dear Mr. Gaffney:

We have purchased a Motorola EXORciser development system for developing 6809 based products. We contracted to an outside vendor for the initial software development on a new product. All software was written in HP64000 Pascal. We will be doing all software maintenance and enhancement in house and we are reluctant to do this in assembly language. Therefore, I am attempting to find a Pascal compiler to run on the 6809 EXORciser which is as compatible as possible with the HP Pascal. I would appreciate any help you can give me on this.

We also have a Texas Instruments DS 990 minicomputer with a Pascal compiler which we would like to use for electrical and electronic engineering support and development. Any information on available Pascal electronics packages would be helpful.

Very truly yours,

MILLER Electric Mfg. Co.
Bruce A. Casner
Project Engineer
P.O. Box 1079
Appleton, WI 54912

Open Forum

June 2, 1983

Could you send me any information you have on Apple and JRT Pascal? Also, I would be very appreciative if you could recommend any books for someone who knows BASIC and 6502 Asembler.

Thank you.

Larry Houston
169 West 8th Street
Peru, IN 46970

December 10, 1982

Mr. Gaffney:

I read about *Pascal News* from a UNIX NetNews (a network of UNIX installations sharing news) article. Could you send me the back editions containing the Lisp compiler-interpreter (written in Pascal)? Enclosed is $15 for the year of back issues containing the Lisp compiler. Send me a bill for shipping if $15 does not cover it all. I am glad to see you continuing this magazine.

Respectfully,

Fred R. Finster
8549 Evanston Ave. N.
Seattle, WA 98103

December 20, 1982

Dear Charles Gaffney (Charlie),

I have been infomed that PUG (Australia) has distributed its last issue (#24) of *Pascal News*, and that the subscription list and balance of funds has been transferred to the U.S.

According to my records, I was paid up through 1984 (renewed for 3 years mid-1981).

However, I understand that there will probably need to be an adjustment, so please could you apply my outstanding subscription toward whatever extension is appropriate.

Please, if possible, inform me what the final position is; I am very keen to maintain my continuity of membership, as I think PUG is very worthwhile.

Thanks a lot.

Yours sincerely,

G. A. Foster
5/138 Clarence Road
Indooroopilly
Queensland, Australia 4068

February 15, 1983

Dear Mr. Shaw:

I know that your job keeping going the PUG is a great one. As we create the Mexican Wang Users Group with 200 members now, after 4 years about 4 people do the whole work.

Our company with 32 people has an obligation to use an accumulative half hour daily to do some investigation; that is why we are interested in implementing the Pascal in our machine, a Want 2200-VS-80.

I believe that our specifications were wrong when we asked for the Portable Pascal P4, because we have not been able to get started.

Dr. Niklaus Wirth wrote me that the PUG has an 360 IBM compiler. I would like to know how can we be able to get it, because our computer with very little modification can run IBM assembler programs.

If I can do anything to help the PUG please let me know.

Thanking you in advance for all the trouble I may cause, looking forward to your answer.

Sincerely,

Miguel M. Soriano Lopez
Technical Director
Data, S.A.
Av. Homero No. 1425-1201
Mexico 5, D.F.

June 5, 1981

Dear Mr. Soriano,

It was a pleasure to hear from you after so many years. I fondly remember that stay in Mexico in 1963.

I guess the best way to "get in touch" with me is by writing, as you did. However, I do not see a chance for me to visit Mexico in the near future, as I am quite committed and busy at my position here at ETH.

Good luck for your Pascal compiler project. Are you aware of the compiler for the 360, also available from PUG? Perhaps it would be easier to use that compiler instead of the P, because your machine is—as you write—similar to the 360.

Sincerely yours,

Prof. Niklaus Wirth
Eidgenossische Technische Hochschule
Instit fur Informatik
ETH-Zentrum
CH-8092 Zurich

August 31, 1983

Dear Charlie:

I use Pascal at the National University of Mexico in a Burroughs 7500 or on a PDP-11, and at my work I am trying to install the Pascal-P you send me last year.

Several doubts I had, I asked Dr. Wirth, who answer me and recommended that will be easy to implement the IBM-370 version, which is more similar to my Wang 2200-VS-80 machine.

On the 25th *Pascal News* is a report about an IBM-370 Pascal. I will like to know if Joseph A. Minor of Cornell Computer Services would like to work together with me, to give to the PUG a Pascal compiler for the Wang VS. Only in Mexico there are more than 200 installations; I believe that at the USA are several hundreds who may be Pascalers, if the PUG will have it.

I hope to hear from you soon, thanking you for the trouble I may cause you.

Sincerely,

Miguel M. Soriano Lopez

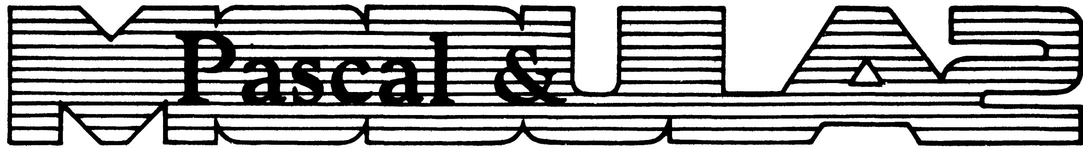
May 26, 1983

Dear Editor:

I would appreciate it if you would publish the enclosed **announcement** of the availability of the Edison System Report entitled "Programming a Personal Computer" in the *Pascal News*.

Yours sincerely,

Per Brinch Hansen
Henry Salvatori Professor of Computer Science
University of Southern California
University Park
Los Angeles, CA 90089



Press Release

FOR IMMEDIATE RELEASE: October 11, 1983

Cleveland, Ohio: The ten year old publication, Pascal News is changing its format and name to Pascal and Modula 2, according to publisher, Charles Gaffney.

For those unfamiliar with computer terminology, Pascal is a small and general purpose computer programming language, originally designed 10 years ago by Professor Niklaus Wirth of Switzerland as a teaching language. Because Pascal is easy to learn and read, and can be efficiently translated by computers, it was adapted for use in business. However, it does have certain restrictions. To meet the increased demand for an all purpose programming language, Wirth designed Modula 2. The structure of Pascal is included in Modula 2, affording simple and quick transition for programmers.

Pascal News was origninally established to be a forum of correspondence for the Pascal Users Group (PUG). Because of the close linkage between these languages, and the rapid growth in the day to day use of computers, Gaffney expanded the publication to include articles and correspondence about Modula 2.

"Modula 2 is a newer language on the cutting edge of computer science," said Gaffney. "Its design allows Pascal to settle into original standards, and removes the pressure to be all things to all people."

Modula 2, sold only under license, will be protected from incompatible revision. The new availability of Modula 2 from at least 4 vendors demands a forum for new users as well as Pascal users. Pascal and Modula 2, sponsoring the Pascal Users Group and organizing the Modula 2 Users Group, will provide that forum.

Pascal News has over 4,000 subscribers in 41 countries, according to Gaffney who is confident the new publication will continue to serve these user groups. Pascal and Modula 2 will provide application software, software tools, articles on programming philosophy, the use of Pascal as a teaching tool, the promotion and application of each language, as well as an important open forum where users contribute informal correspondence of general interest to the group.

P.O. BOX 538, CHESTERLAND, OHIO 44026 • (216)729-3227

Two Pascal Devices

By Harley Flanders

Florida Atlantic University

1. THE FORWARD DECLARATION

It was brought to my attention by H. S. Wilf that the reserved word **forward** is not necessarily included in all versions of Pascal. If we examine Jensen and Wirth[1], we find **forward** discussed on page 82 of the ***User Manual***, but not in Appendix C, *Syntax*, pages 110-115, nor in the railroad diagrams, pages 116-118; however, in Appendix E, *Error Number Summary,* page 120, while nowhere in the ***Report***, pages 133-167.

Suppose we have a program in which several procedures call each other recursively. The usual way to handle this is via a series of **forward** declarations. It is possible to accomplish the same thing without using **forward** at all. Suppose, for example, that we have the declarations

```
procedure B; forward;

procedure C; forward;

Procedure A;
   <declarations>
   begin
   <statements A1>; B; <statements A2>
   end;

procedure B;
   <declarations>
   begin
   <statements B1>; C; <statements B2>
   end;

procedure C;
   <declarations>
   begin
   <statements C1>; A; <statements C2>
   end;
```

The following single procedure declaration does the same.

```
var CONTROL: Char;

procedure ABC;
   <declarations>
   begin
   case CONTROL of
     'A': begin <statements A1>;
          CONTROL := B; ABC;
          <statements A2> end;
     'B': begin <statements B1>;
          CONTROL := C; ABC;
          <statements B2> end;
     'C': begin <statements C1>;
          CONTROL := A; ABC;
          <statements C2> end
     end   { case }
   end;   { ABC }
```

The statement calling ABC must initialize CONTROL to the first entry value. Clearly, many variations on this theme are possible.

2. THE EXIT PROCEDURE

UCSD Pascal restricts the **goto** statement so it only allows jumps within a block. Hence **goto** cannot be used to exit a nested sequence of procedures. The Exit procedure was introduced into UCSD to make up for this shortcoming; however, it fails to do the job if the nested sequence happens to include recursive calls of a procedure. This can be quite inconvenient at times. Suppose, for example, one is searching an array, and the search proceeds by testing an element then, if unsuccessful, it partitions the array and tests the pieces recursively. Of course, once the sought element is found, the search should be stopped. But the search procedure may be deeply nested at that time.

A clumsy way out is to introduce a Boolean variable FOUND and rewrite the body of the search procedure as follows:

```
procedure SEARCH;
  <declarations>
  begin
  if not FOUND then <statements of SEARCH>
  end;
```

The (external) call of SEARCH must be replaced by

FOUND := False; SEARCH

This is costly because it adds an extra test to each call of SEARCH. The following is an alternative using Exit. Assume that we are searching for X and that A denotes a test value.

```
procedure DUMMY;
   procedure SEARCH;
     begin
     <statements of SEARCH>
     if A = X then Exit(DUMMY) else ... SEARCH
     end;
   begin SEARCH end;
```

Remember the UCSD rule is that Exit (PROC) is a jump to immediately after the **most recent** call of PROC, passing through all more recently called procedures along the way, and doing some incidental housekeeping, like closing files those procedures opened.

Reference

1. K. Jensen and N. Wirth, ***Pascal User Manual and Report,*** 2/e, Springer-Verlag, 1978.

"Zuse"

A Compiler Writer

ZUSE USER'S MANUAL
Unix Implementation
Version 1.0

by
Arthur Pyster

Department of Computer Science
University of California at Santa Barbara
Santa Barbara, CA 93106

copyright (c): Regents of the University
of California, May 1981

This work was in part supported by a grant from the Instructional Development Program of UC Santa Barbara.

1.0 Introduction

1.1 Background

Zuse is a translator writing system written in Pascal which produces translators which are themselves written in Pascal. It is quite simple to use and alleviates much of the tedium inherent in writing a translator from scratch. It is named after Konrad Zuse whose visionary work on the programming language Plankalkul in the mid 1940s should be an inspiration to everyone.

This user's manual assumes that you are already familiar with the principles of translator writing and syntax-directed translation. Such terms as "BNF grammar" and "LL(1) parser" are used freely without explanation. If you lack this background, you should refer to one of the standard texts on translator writing (Aho and Ullman, "Principles of Compiler Design", Addison-Wesley 1977; Lewis, Rosenkrantz, and Stearns, "Compiler Design Theory", Addison-Wesley 1976; Pyster, "Compiler Design and Construction", Van Nostrand Reinhold 1980).

Zuse has been designed to be highly portable across different Pascal implementations. Only a handful of lines of code have been written using implementation-dependent features; e.g., Zuse presumes that type char is the ascii character set. This manual describes Zuse as it is implemented on Unix. A separate document describing any deviations from this manual should be obtained from whoever is responsible for Zuse's installation at your computer center.

1.2 How to use Zuse

Zuse actually consists of two programs: generate.o - an executable program which accepts a translation grammar as input and generates several files which will be needed for the translator eventually produced; and skeleton.p - a partial translator written in Pascal which must be augmented by files produced both by you and by generate.o in order to become a complete Pascal program which can be compiled. Figure 1 shows the creation of a translator, skeleton.o. Figure 2 shows the execution of that translator to translate source string x. When creating a translator, all of the files except for the user-defined translation grammar, and LLsup.i are part of or generated by Zuse. When executing the translator, only the source string which is to be translated needs to be provided by the user. LLgram is produced by Zuse. The particular manner in which Zuse creates the necessary files is described in later sections of this manual.

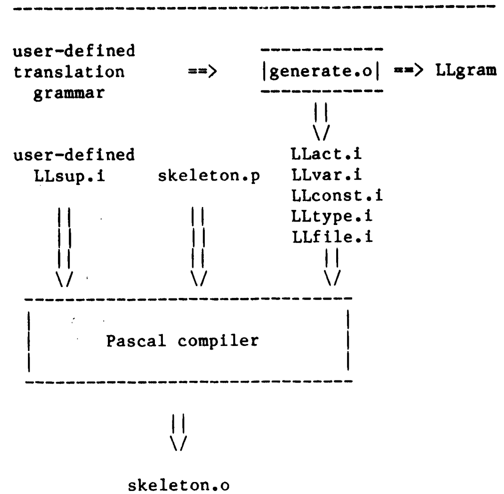


Fig. 1 Creating a Translator

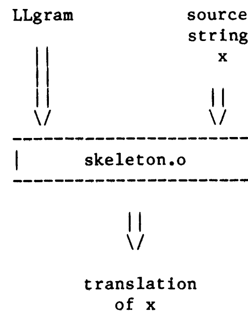


Fig. 2 Executing a Translator

Zuse's two programs are used in the sequence listed below to create a translator:

1) Write a context-free grammar which specifies the syntax of the source language. The grammar should be prepared as a file using any text editor. It can be stored under any file name. Because of the parsing method supported by Zuse, the grammar must be an extended LL(1) grammar. The permissible extensions are specified in later sections. This grammar will eventually be used to produce a top-down left-to-right parser.

2) Embed action code into the grammar which specifies the steps to be taken by the translator during parsing. A language definition with both a syntactic specification and action code is referred to as a "translation grammar". The translation grammar specifies the actions to be taken during the parsing of a string. These actions produce a translation of the source string.

3) Prepare Pascal code which defines all supporting routines which will be called by the action code, including the lexical analyzer -- LLNextToken. These routines, which will be needed when skeleton.p is compiled, should be stored in the file LLsup.i.

4) Execute generate.o on the translation grammar just prepared. Generate.o expects the grammar to come from the standard input file so you must redirect your file:

```
generate.o < MyFile
```

Any errors it uncovers will be reported on the standard output file. Generate.o creates several Pascal text files which must be embedded into skeleton.p along with LLsup.i. Skeleton.p has been peppered with "include" directives so that these files will automatically be included during its compilation.

One additional file will be created - LLgram. It is a modified version of the grammar specified in step 2, which has been compressed to facilitate use by skeleton.o, the compiled form of skeleton.p. Skeleton.o reads LLgram before translation begins.

5) Compile skeleton.p. If you are using the Pascal compiler "pi", then just type:

```
pi skeleton.p; mv obj skeleton.o
```

If you have another Pascal compiler, then invoke it in the standard way. Assuming there were no errors in the specifica-

tion of the grammar and the definition of the routines in LLsup.i, the resulting object code, skeleton.o, will be a working translator. If there are errors in the grammar, then all the steps just listed will have to be repeated. If the translation grammar is correct, but the support routines are incorrect, then they must be corrected and skeleton.p must be recompiled. You must repeat this process until you are satisfied that the translator is correct.

6) Add error processing capabilities. Skeleton.o will detect any syntactic error in a source string. The detection and processing of semantic errors is left entirely in your hands as the translator writer. Skeleton.o's default error recovery is to terminate the translation, causing an appropriate message to be printed. Zuse also supports optional sophisticated error processing facilities which allow you to specify a number of possible recoveries when a parsing error is detected. These are explained in section 7.

1.3 Manual organization

This user manual itself is organized into sections based on the steps just listed in section 1.2. A running example is used throughout to illustrate concepts as they are introduced.

For further information about Zuse, the reader is urged to contact the author at the Department of Computer Science, University of California, Santa Barbara, California 93106, phone: (805) 961-3236 or x-4321.

2.0 Write A Context-Free Grammar

The first step in generating a translator is to prepare a context-free grammar which specifies the syntax of the language to be translated. A context-free grammar G classically has four components:

- nonterminals - the grammatical categories
- terminals - the alphabet of the language
- axiom - a nonterminal which begins all derivations
- productions - the rewriting rules

In fact, Zuse uses a somewhat different structure dictated by the need to distinguish between different types of terminals. Two classes of terminals will be defined: "groups" and "literals". Details about them are presented in section 2.3.

The grammar specification is divided into declarations and productions. Each vocabulary symbol used in the grammar must be declared, indicating what type of symbol it is. Just as in Pascal, a symbol must be declared before it can be referenced. But before describing how to write the grammar, the language which will be the basis for a running example throughout the manual will be described.

2.1 The language EXPRESSIONS

At this point the language which will serve as the running example throughout the manual will be introduced. The language EXPRESSIONS contains possibly empty sequences of arithmetic expressions terminated by semicolons. Its grammar is called G_EXPRESSIONS. The operators are four basic arithmetic operations on integer values:

```
+ - * /
```

Operands are unsigned integer literals. Some sample strings of EXPRESSIONS are:

```
(3+4)*(5-6); 4-6; 4 / 6;

12 / 3;

3;
```

The arithmetic operators follow common precedence rules; i.e., * and / are performed before + and -, with operations being performed left to right within precedence. Parenthesization can override any default precedence.

The translation of a member of EXPRESSIONS will be its numeric value. Hence, the translator in this case will be an expression interpreter. For the three examples above, the translations are:

```
-7  -2  0

4

3
```

You will type in an expression from the terminal, and your interpreter will print its value immediately below your input. If you type an invalid expression, then an appropriate error message will be printed just below the incorrect input. Hence, in this case there really is no "object code" for the translation, since the display of the expression's value on the terminal is the only desired output.

2.2 Lexical structure of Zuse grammars

2.2.1 Tokens

Zuse grammars have much the same lexical structure as Pascal programs. Within a grammar a "token" is a sequence of printable characters which has no embedded blanks, tabs, or newlines. In certain cases it may be necessary to surround a token with single quotes:

```
' .. token .. '
```

because the unquoted token is part of the metalanguage used by Zuse to specify the grammar. For example, each production ends with a semicolon. Hence, it is impossible to have a literal semicolon as a terminal symbol on the right-hand side of a production unless it is surrounded by quotes:

```
X = ... ';' ...    ;
```

In other contexts where there is no ambiguity, the semicolon can be used without quotes.

Tokens may start in any column. Blanks, tabs, and newlines are token separators. Blank lines may be freely inserted anywhere in a grammar. For convenience in later discussion, the word "spacer" will be uniformly used to mean any positive number of blanks, tabs, and newlines.

Zuse is sensitive to upper and lower case letters. For example, the following three symbols could all be declared as distinct nonterminals in your grammar:

```
PROGRAM  program  PrOgRaM
```

so be very careful, especially if your Pascal compiler does not make such distinctions for identifiers declared in Pascal programs. Even if your compiler would treat the three symbols written above as the same identifier, Zuse will not.

The maximum permissible length of a symbol is 12. If you write a symbol longer than the permitted maximum, generate.o will print an error message and disgard all characters of that symbol beyond the maximum.

2.2.2 Comments

A comment may be inserted anywhere a spacer is allowed. In Zuse comments have the form:

```
(*  .. comment .. *)
```

This is one of the two formats for comments used in Pascal. The other form of Pascal comment, { .. }, is not allowed in Zuse because the curly brackets are used for other purposes as described later.

2.3 Declarations

Every symbol used in the grammar must be declared, although the order of declaration is not significant. A symbol can only be declared once. Zuse will tell you if you declare a symbol more than once or reference an undeclared symbol in a production.

The same basic format is used for all declarations:

```
%SPECIFIER  LIST_OF_SYMBOLS
```

The SPECIFIER, which is part of the same token as '%', and so cannot be separated from it by a spacer, indicates the type of component being declared:

n N - nonterminal

a A - axiom

l L - literal

g G - group

Note that either lower or upper case letters can be used. There are three other specifiers used for declaring other types of symbols, which are described later.

2.3.1 Terminals

Terminal symbols represent the lexemes of the source string. They are divided into two kinds -- "literals" and "groups". For example, in a grammar for Pascal, we might find terminals declared for the keywords such as "do", "program", "begin", and "var", for operators such as ":=", "<>", and "=", and for the identifiers and integers. Any sequence of characters with no embedded spacers can be declared as either a literal or a group.

A literal represents itself; i.e., it is a symbol which literally appears in a source string. The terminals "<>", "var", and "=" are examples of literals. Reading a Pascal program, we would expect to find literal occurrences of these symbols. The other form of terminal is a group, which represents a collection of lexically related symbols such as the integers or identifiers.

The grammar, G_EXPRESSION, has seven literal symbols which can be declared by:

```
%l + - * / ( ) ;
```

The order in which the literals are declared is not important, but every literal which will be used in a production must be declared.

The "%l" could appear anywhere on the line ("%L" could have been used instead, but not "% l" since no embedded spacers are allowed), followed by the list of literal symbols. Elements of the list are separated by one or more spacers.

Literals can be declared over several lines, using any number of separate literal declarations. The sequence of declarations:

```
%l +        (* additive ops *)
   -

%l *        (* multiplicative ops *)
%l /

%l (
%l ) %l ;   (* note ";" doesn't need
               quotes here *)
```

is equivalent to the earlier declaration.

At the translator-writer's discretion, the lexical analyzer can group symbols into a syntactic category rather than relying on the grammar to do so directly. Doing so can sometimes simplify the grammar significantly; this is usually true for integers. Since EXPRESSIONS contains positive integers, they will be used to illustrate the concept of a lexical group.

There is such a large number of distinct integers that it is impossible to enumerate them, yet they can certainly be considered indivisible lexical units. To make it possible to write a grammar which does not detail how integers are structured and does not attempt the impossible task of enumerating them, the notion of a token group is introduced. The entire set of integers can be declared by:

```
%g INTEGER
```

Having done so, you can use the symbol INTEGER in productions to stand for any integer wherever a terminal symbol is legal. The lexical analyzer LLNextToken must contain logic to recognize and classify integers within the source string. The classification details are discussed in section 5.

It is important to realize that the word "INTEGER" used in the above declaration has no special meaning to Zuse. The symbol "gzorp" could just as easily have been used:

```
%g gzorp   (* stands for the integers *)
```

"INTEGER" was chosen because it is a good mnemonic, but other equally good (but distinct) mnemonics such as "integer", "int", and "Integer" could have been selected instead.

2.3.2 Axiom

In addition to declaring the terminal symbols, you must also declare the nonterminals and indicate which nonterminal is the grammar axiom. The axiom of G_EXPRESSIONS, "Ax" is declared by:

```
%a Ax
```

or

```
%A Ax
```

This declaration can appear anywhere in the declaration section. It also has the effect of declaring "Ax" to be a nonterminal, so it should not be redeclared as a nonterminal elsewhere. If you forget to declare an axiom, generate.o will remind you.

2.3.3 Nonterminals

A nonterminal is any token containing no embedded spacer. They are declared using the specifier 'n' (or 'N'):

```
%n Asop Mdop E T E-list P T-list
```

Nonterminal names are separated by one or more spacers. Note that nonterminals E-list and T-list include a hyphen, making them illegal Pascal identifiers, but perfectly valid nonterminals. All nonterminals are shown here on a single line, but they could just as well have been written over several lines as was done for the grammar literals.

2.3.4 Terminating the declarations

When all declarations have been stated, the declaration section is terminated by '%%'. Hence, the entire declaration section for G_EXPRESSIONS could look like:

```
%a Ax
%n Asop Mdop E T E-list P T-list
%g INTEGER
%l + - * / ( ) ;
%%
```

2.4 Productions

2.4.1 Alternative productions

Once the declaration section is complete, the grammar productions must be specified. They follow the "%%" which ends the declaration section. All productions with the same left-hand side, called alternative productions, must be declared together. The format for declaring the set of alternative productions for some nonterminal X is:

```
X = LIST_OF_SYMBOLS  ;

  = LIST_OF_SYMBOLS  ;
    ...
  = LIST_OF_SYMBOLS  ;
```

where "=" separates the left and right-hand sides of the production, LIST_OF_SYMBOLS is a possibly empty sequence of vocabulary symbols, and ";" terminates each alternative production. All nonterminals, literals, and groups used in a production must appear in a declaration. Members of the list of symbols are separated by one or more spacers.

The productions for G_EXPRESSIONS are:

```
Ax =  ;  (* empty production *)
   =  E ';' Ax ;  (* note the use of a quoted ; *)
```

```
E = T E-list ;

E-list = Asop T E-list ;
       = ;  (* another empty production *)

T = P T-list ;

T-list = Mdop P T-list ;
       = ;

P = ( E ) ;
  = INTEGER ;

Asop = + ;
     = - ;

Mdop = * ;
     = / ;
```

Note the use of both literals and groups in the symbol lists on the right-hand side of productions and the use of the quoted semicolon in the second alternative production for "Ax". The literal semicolon must be quoted to distinguish it from the end of the production marker. On the other hand, the declaration of the literal semicolon did not require quotes because a semicolon has no special significance in declarations.

2.4.2 Selection set conflicts

Because Zuse is based on LL(1) parsing, the selection set of each production must be computed. This tells skeleton.o which alternative production to select when it needs to expand a nonterminal while building the parse tree top-down for a source string. Generate.o will compute the selection set of each production for you. This computation is very tedious and error-prone when done manually, and its automatic computation is one of the more pleasant features of Zuse.

The grammar for G_EXPRESSIONS given above is LL(1); i.e., there are no conflicts in the computed selection sets of alternative productions. Sometimes, however, you will write a grammar in which the selection sets of two alternative productions are not disjoint. Generate.o will always inform you when this occurs by printing a message stating which alternatives have the conflict, and which selection set element they share. The message will appear on the standard output device during the processing of the grammar. However, even if there are conflicts, generate.o will still produce a valid parser. It uses a default tie-breaking strategy whenever conflicts arise: If two alternative productions have a selection set element in common, the production which appears first in the grammar will be selected. The conflicting token will be erased from the selection set of the later occurring alternative.

This tie-breaking strategy is fine provided it is consistent with how you envisioned the parse to proceed. However, Zuse provides an easy way to override the default tie-breaker whenever the default choice is inappropriate. This is nicely illustrated using the classic ambiguity involving the optional "else" clause of an if-statement.

The ambiguity arises when considering nested if-statements:

```
if p then if r then s else t
```

can be parsed so that the else-clause is associated with either the first or the second if-statement, as reflected by the two physical nestings:

```
if p then                (1)
   if r then s
   else t
```

or

```
if p then                (2)
   if r then s
else t
```

The first nesting is common to most languages including Pascal. The problem is getting the parser to understand which nesting is meant.

The ambiguity is reflected in a grammar which is not quite LL(1). Two alternative productions both contain the literal 'else':

```
IF_STMT = if PRED then STMT ELSE_PT  ;

ELSE_PT = else STMT ;
        = ;
```

The selection set of each alternative of ELSE_PT contains "else". The presence of other tokens in these selection sets would depend on the other features of the language in which this statement is embedded. According to the default tie-breaking strategy of Zuse, the first alternative would be used. This corresponds to (1) above.

To illustrate the statement of specific selection set elements, suppose that the order of the two alternatives were reversed. To achieve the same nesting, you would have to write:

```
ELSE_PT =  ;
        =  else STMT %else ;  (* choose this prod *)
```

A specific selection set element can be written as part of a production. This list of terminals, begun by "%", and separated by one or more spacers appears just before the semicolon ending the production. Specifying "% else" tells skeleton.o to use the second alternative no matter which other alternative productions might also have "else" in their selection sets. Any production can be written with a selection set, although no two alternatives should specify the same selection set element.

Since you can specify selection set elements, you must also have a way to indicate which production to select when the end of the source string has been reached. None of the declared terminal symbols is appropriate. "@" is a pre-defined group used in a grammar to signify "end of source". It can only be used in a selection set and in synchronization specifications (section 7.2), never as an ordinary terminal on the right-hand side of a production. Note that since "@" means "end of source", it should not be used in other contexts. If you have a language in which "@" is a token, then declare a group such as "AT-SIGN" and have the lexical analyzer associate "@" with "AT-SIGN".

Occassionally you may want to force the selection of a production no matter what the next token's value really is. This is normally used for error processing in which a production really contains nothing more than a specification of the error recovery strategy you want to employ. Forcing the selection of a production will ensure that the desired error recovery strategy is adopted. Otherwise, if the error occurred in the token which would have caused you to select the production with the recovery strategy, that strategy will not be applied. This is discussed in more detail in the section 7 on error processing.

In order to force the selection of a production, a predefined group element "any" is introduced. (All letters of "any" must be lower case.) Like "@", it should only be used in selection sets and in specifying synchronization. It is used to force the selection of a production from among a set of alternatives.

Suppose the current token is w and "any" is in the selection set of production p. If no previously occurring alternative to p has w in its selection set, then p will be selected, even if the current token is illegal.

2.4.3 LLselect

In a complex grammar it may be hard for you to anticipate what the selection set of particular productions will be. To help you debug your grammar, as well as for documentation, generate.o will optionally produce a reformatted grammar for you in file "LLselect". The reformatted grammar will be produced if you specify "%s" as a declaration anywhere in the declaration section. The reformatted grammar will contain the productions numbered in increasing order by line number from the original grammar. All error processing code will have been removed for easier readability, and all action code will have been replaced by the string "{a..}". Most important is the addition of the entire selection set of each alternative after all conflicts have been resolved. The messages stating selection set conflicts which were reported on the standard output during the execution of generate.o will be included at the front of LLselect. Appendix A contains LLselect for G_EXPRESSIONS and Appendix B contains LLselect for a preprocesor for FORTRAN.

2.4.4 Terminating the productions

The grammar productions are terminated in a manner similar to the way the declarations are, by writing "%%" where the beginning of a production is expected.

3.0 Embed Action Routines

3.1 Format

Once the grammar has been defined, you must add action code which specifies how the translation will proceed. Although this step can be done while the grammar is being written, it is often more convenient and more reliable to first develop the grammar, embedding action code only after the grammar is complete.

Action code may appear anywhere among the list of vocabulary symbols on the right-hand side of a production. The code is actually a sequence of Pascal statements enclosed by curly brackets:

'{a' Pascal statements '}'

The first character after '{' must be an 'a' to indicate that it is action code. Other characters, discussed in section 7, are used for other types of code.

Rewriting the grammar productions for G_EXPRESSIONS to include action code yields:

```
Ax =  ;   (* empty production *)

    = {a init} E {a writeln(popopand);}
      ';' Ax ;  (* note the use of a quoted ; *)

E = T E-list ;

E-list = Asop {a pushoptor($1.operator)} T
            {a r := popopand;
               l := popopand;
               popoptor(op);
               if op = '+' then
                  pushopand(l+r)
               else
                  pushopand(l-r)}
         E-list ;

       =  ;    (* another empty production *)

T = P T-list ;

T-list = Mdop {a pushoptor($1.operator)} P
            {a r := popopand;
               l := popopand;
               popoptor(op);
               if op = '*' then
                  pushopand(l*r)
               else
                  pushopand(l div r)}

         T-list ;

       = ;

P = ( E ) ;
  = INTEGER {a pushopand($1.operand)} ;

Asop = + {a $0.operator := '+'};
     = - {a $0.operator := '-'};

Mdop = * {a $0.operator := '*'};
     = / {a $0.operator := '/'};
```

Several aspects of this revised grammar warrant explanation. For the moment ignore those strange looking identifiers which begin with "$". They refer to special variables which will be explained in section 3.5.

Action code specifies how the translation is to take place. All aspects of that specification are left in your hands. You can either write all of your code directly in the grammar or you can call separately declared procedures and functions from within the action code, or mix the two styles. The grammar above is written in a mixed style. The action code in the first alternative for T-list has two assignment statements followed by a procedure call, followed by an if-then-else statement. The action code for the second alternative of P has just a single procedure call. You must decide which support routines (as those user-defined routines called from action code are called) to define and what they will do. G_EXPRESSIONS has five support routines referenced in the action code:

init pushopand popopand pushoptor popoptor

The declaration of these five routines must be included in LLsup.i, which contains the support routines, before skeleton.p can be compiled. However, at this point you only need to know what these routines do so that you can properly write the action code.

The action code will map the infix expressions into two stacks -- "opandstk" (operand stack) and "optorstk" (operator stack) -- where the operands and operators of the expression will be held, respectively. The algorithm to evaluate infix expressions using two stacks is quite standard, and it is assumed that you are familiar with it. The code defining these five functions is specified in section 4. "Init" initializes the two arrays which represent the stacks and two integer variables, "topopand" and "topoptor", which point to the top of "opandstk" and "optorstk", respectively. "Popopand" and "pushopand" pop and push elements onto opandstk, while "popoptor" and "pushoptor" are the analogous routines for optorstk.

3.2 Variable, constant, type, and file declarations

3.2.1 Variable declarations

Two variables are referenced in the action code -- "l" and "r". These variables hold temporary values of the left and right operands of some operator. Because Pascal requires the declaration of each variable which is referenced, there must be some provision for declaring these variables. Note that Zuse itself cannot have already declared them because the particular set of variables needed for the action routines will vary from translator to translator. To permit user-declared variables, an additional declaration type is permitted in grammars in the declaration section:

%v Pascal variable declarations

If a 'v' (or 'V') specifier is used in a declaration, then the variable declaration is copied verbatim into file "LLvar.i". Note that each declaration ends with a semicolon. If more than one variable is declared, they are copied in the order in which they appear in the grammar.

G_EXPRESSIONS needs several declarations:

```
%v op: char;  (* the operator popped from
                 optorstk *)

%v toptorstk: integer; (* top of optorstk *)
   topandstk: integer; (* top of opandstk *)

%v opandstk: array[1..stksize] of integer;
   optorstk: array[1..stksize] of char;

%v l,r: integer;  (* temps *)
```

These six variable declarations can appear anywhere in the declaration section. Since the order of variable declarations is not important in Pascal, they can be declared in any order. Note that several variables can be declared using a single "%v" as in the declaration of toptorstk and topandstk.

3.2.2 Constant declarations

The integer constant "stksize" is referenced in the declaration of opandstk and optorstk above. This constant and any others which you need for your action code can be specified through a constant declaration in the grammar. A 'c' or 'C' is used to specify a constant declaration. G_EXPRESSIONS needs:

```
%c stksize = 20;  (* maximum depth of stk *)
```

Constant declarations will be placed into the file "LLconst.i" for inclusion into skeleton.p. They will appear in LLconst.i in the same order as they appear in your grammar. It is your responsibility to guarantee that a constant is defined before it is referenced.

3.2.3 Type declarations

Although they are not needed for this example, you can declare new types using the 't' or 'T' specifier. For example, if we wanted our interpretor to only work on nonnegative numbers, we might declare:

```
%t NonNegative = 0..maxint;
```

and substitute NonNegative for integer in the variable declarations:

```
   ...
   %v toptorstk: NonNegative; (* top of optorstk *)
      topandstk: NonNegative; (* top of opandstk *)
   ...
```

Type declarations will be placed into the file "LLtype.i" for inclusion into skeleton.p. They will appear in LLtype.i in the same order as they appear in your grammar. It is your responsibility to guarantee that a type is defined before it is referenced.

3.2.4 File declarations

The purpose of skeleton.o is either to interpret the input or to map it into some object code. It therefore must have a file from which the source string is read and, in the latter case, a file where the object code is placed. In addition, a complex compiler may also need several other files. The file declarations themselves can be handled as ordinary variable declarations. For example, if we wanted a to write a C compiler, the object code could be placed in a file called "object" which was declared by:

```
%v  object: file of char;
```

The action code which produced the object code would write tc this file.

An additional problem arises, however, because Pascal also requires that each file be listed in the program statement. Hence a %f declaration is introduced. The program heading fo skeleton.p has the form:

```
     program skeleton( input, output, LLgram
#include LLfile.i
                          );
```

If you wish to augment skeleton.p with new files, their names should be listed in a %f declaration, begun by a comma, and separated by commas. If we wanted to add an object and a message file to skeleton.p we would declare these variables by:

```
     %v object: file of char;
        message: file of char;
```

and also declare the files by:

```
     %f ,object, message
```

The former would be included into the variable declaration section of skeleton.p, while the latter would yield the program statement:

```
     program skeleton( input, output, LLgram
,object, message
                          );
```

3.3 Naming conventions

Because Pascal is a block-structured language, the scope of declarations in skeleton.p is very important. You might accidentally select a name for one of your variables, types, constants, or support routines which already has been declared in skeleton.p at the same lexical level. This would cause a compile-time error when you tried to compile skeleton.p. To minimize the risk of such collisions all identifers in skeleton.p which are at the same lexical level as those identifiers which you will declare for inclusion within it begin with the characters "LL". For example, the main procedure of skeleton.p is called "LLMain". If you simply avoid declaring identifiers which begin with "LL" you should never encounter any difficulties.

3.4 Parsing action

Skeleton.o will construct a parse tree for the source string in a top-down left-to-right manner. The axiom you declared in the grammar will be the root of the parse tree. It will compare the first token of the source string against the selection sets of the alternative productions for the axiom. Assuming it finds a production with the required selection set element, it will expand the axiom by hanging tree nodes from it corresponding to the right-hand of the selected production. It will then examine the leftmost child of the axiom which was just added. There are three types of nodes that child can be, depending on the kind of symbol from the right-hand side of a production which it stands for -- nonterminal, terminal, and action. The translator's response will depend on which type of node it finds.

If the child is a nonterminal node, the same process which was just applied to the axiom will be repeated for the child. The alternative productions of the nonterminal will be scanned to find one whose selection set matches the current token. Failure to find such an alternative indicates that the source string has a syntactic error, and error processing as detailed in section 7 will be initiated.

On the other hand, if the child is a terminal, the parser will compare the first token of the source string against that terminal. If they "match"; i.e, they are equal, the parser will advance to the right sibling of that node, and advance to the next token in the source string. At this point the whole process will be repeated with the particular action taken depending on whether the tree node is a nonterminal action code, or a terminal.

If the child is action code, that code will be executed. Presumably this code will manipulate the variables declared in the grammar by "%v" declarations in order to effect the desired translation. Once this code completes execution, the parser will advance to the right sibling of that tree node, but will not advance to the next token in the source string since nothing in the parse tree has been "matched" against it.

When the right-most child of a parent nonterminal has been visited in the manner just described, the parsing of that nonterminal is considered complete. Parsing continues with the right sibling of that parent node. This process iterates until an error is uncovered, in which case the error recovery policy dictates what then happens, or until the end of the source string is reached, or until the entire parse tree has been constructed. If the end of the source string is reached before the entire parse tree has been constructed or conversely, the string is not in the source language and error processing is initiated.

The order in which nodes are added to the tree and then examined dictates when action code will be executed. For example, in G_EXPRESSIONS if the second production is selected when expanding axiom Ax, then the first thing the parser will do is call init, the action routine which initializes the data structures necessary to compute the value of the expression. Once initialization is complete, the parser will attempt to expand E into a complete expression. Based on the other action code which will be executed during that expansion, the value of the expression will be on the top of opandstk when E has been completely expanded. At that point other action code is executed which causes opandstk to be popped, and the value to be printed. The parser then moves on to the semicolon, and finally to Ax. If this interpreter is to operate correctly, the other action code embedded in the remaining productions must ensure that the value of the string derived from E is stored atop opandstk. To understand how this is done, the use of synthesized attributes to pass information throughout the parse tree must be examined.

3.5 Attributes

The last feature of the action code which warrants explanation is the appearance of those strange variable names with a "$" in them. They arise from the use of <u>attributes</u> to pass information through the tree.

Each node of the parse tree has associated with it a variable or "attribute" which can be used within action code to compute the translation of the source string. For G_EXPRESSIONS this attribute will be used to pass information about integers and operators in the source string. Because the use of the attribute will vary so greatly with the source language and its intended translation, it would be awkward to predefine the data type which the attribute has. Therefore, each grammar writer must declare the attribute data type using a type declaration:

```
%t LLattribute = type-declaration
```

The reserved name "LLattribute" must be used for this purpose.

Since any Pascal type declaration can be used, a record can be declared to actually provide several distinct attributes. Hence, the restriction to a single attribute per parse tree node is not actually a hindrence. For example, G_EXPRESSIONS needs

some way to store both integer values and char values, leading to the declaration:

```
%t LLattribute = record
                    operator: char;
                    operand:  integer
                 end;
```

In fact, if you would prefer different nodes to have different attributes, rather than having each node have all attributes, this is readily achieved through a variant record:

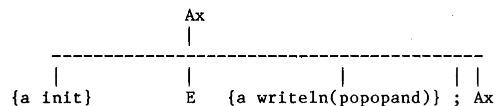
```
%t LLattribute = record
                    case selector-type of
                       selector1: (field1);
                       selector2: (field2);
                          ...
                       selectork: (fieldk)
                 end;
```

In order to refer to an attribute, the action code must use a special naming scheme involving "$". "$" has a special meaning when used inside action code. If n is an unsigned positive integer, then "$n" refers to the attribute of the n-th vocabulary symbol on the right-hand side of the production in which the "$n" appears. "$0" refers to the attribute of the left-hand side of the production. In other contexts, "$" has no special meaning. For example, the first alternative for E-list:

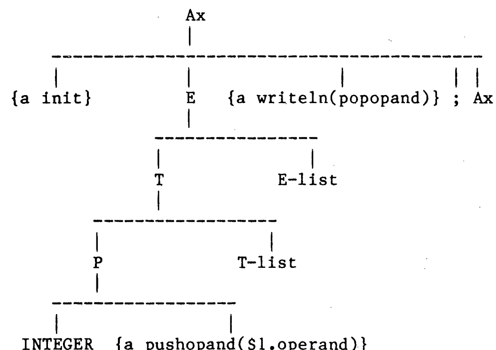
```
E-list = Asop {a pushoptor($1.operator)} T
           {a r := popopand;
              l := popopand;
              popoptor(op);
              if op = '+' then
                 pushopand(l+r)
              else
                 pushopand(l-r);}
        E-list ;
```

contains action code which has "$1.operator" in it. In this case, $1.operator refers to the value of the operator field of the attribute of Asop which is the first symbol on the right-hand side of that production.

To better understand the way in which information is passed up the tree, consider the parsing of "3+4;". The derivation will begin constructing the parse tree:



Init will be called to initialize the data structures, in this case, setting toptorstk and topandstk to be zero showing that no elements have been stacked yet. Then the parser will expand E, T and P in turn giving:

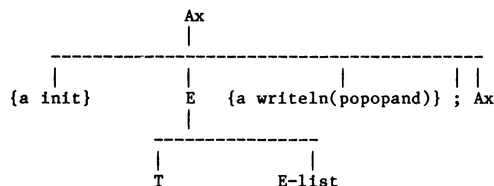


At this point the parser will visit the node labeled INTEGER and match it against the current token "3" in the source string. The lexical analyzer should classify "3" as an INTEGER so that the match will succeed. The current token also has an attribute. The lexical analyzer will assign the integer value 3 to the operand field of that token's attribute.

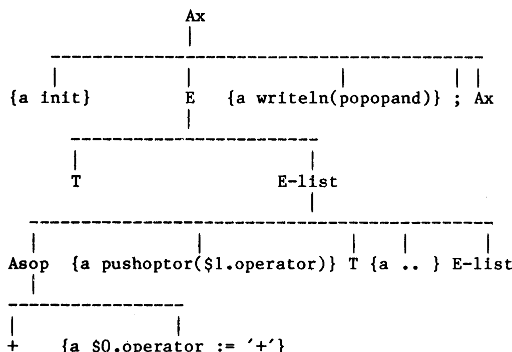
Next the parser advances to visit the action code which is to the right of the node labeled INTEGER. This action code will be executed causing $1.operand to be pushed onto the operand stack. The first vocabulary symbol starting from the left end among its siblings is the node labeled INTEGER which was just matched against "3" in the source string. Hence, the integer value 3 is pushed onto the operand stack.

At this point the right-most child of the node labeled P has been processed. Hence, the parser would next visit the node labeled T-list. The second alternative for T-list will be selected causing T-list to expand into the empty production.

Redrawing the parse tree, eliminating nodes already visited which can play no further role in the translation gives:



E-list and Asop now expand to yield:



The first alternative production is selected for Asop because the current token is "+". The action code in that production assigns $0.operator the character value '+'. $0 refers to the attribute on the left-hand side of the production, which corresponds to the node labeled Asop. When that action code is executed, the operator field of the node labeled Asop is assigned the value '+'.

The next node visited is the action code to the right of Asop. It refers to $1.operator. This is the value just assigned to the operator field of Asop's attribute.

This scenario continues a while longer until the entire parse tree is formed, but by now the basic information passing mechanism using attributes should be clear.

There is a simple restriction on the use of attributes in action code which is dictated by the order in which the parse tree is constructed. An attribute must have a value before it can be referenced in action code. Since skeleton.o parses top-down left-to-right, $n in action code A only has a value if the n-th vocabulary symbol occurs to the left of A. For example:

```
X = Y1 {a .. x := $2 ..} Y2 Y3
```

would be nonsensical because the value of $2; i.e., the attribute of Y2, will not be known when the action code referencing it is executed.

Although the reference to $2 makes no sense in the above production, the very similar:

```
X = Y1 {a .. $2 := x ..} Y2 Y3
```

which assigns a value to $2 is perfectly reasonable. The attribute of Y2 would be assigned a value which could then be passed down the parse tree when Y2 was expanded (assuming it were a non-terminal). This gives you the power of both synthesized and inherited attributes.

In fact, there is no logical reason why you should not be able to assign a value to the attribute of any vocabulary symbol anywhere in the production from any action code in the production. However, generate.o has a restriction on assigning values

to attributes forced by design decisions for generate.o. You can assign a value to the vocabulary symbol to the immediate right of the action code which contains the assignment -- but not to any symbol further to the right. Hence, the production just above is legal, but the similar:

```
X = Y1 {a .. $3 := x ..} Y2 Y3
```

is not legal because the symbol to the immediate right of the action code is the second vocabulary symbol, not the third one.

4.0 Define support routines

The support routines referenced in the action code must be defined before skeleton.p can be compiled. They can freely reference any variables, constants, types, or files declared by the translator-writer in the grammar. For the example, the five routines are:

```
procedure init;
   begin
      topandstk := 0;
      toptorstk := 0;
   end;

function popopand: integer;
   begin
      if topandstk = 0 then begin
         writeln('operand stack underflow');
         LLFatal; {terminate translation}
         end
      else begin
         popopand := opandstk[topandstk];
         topandstk := topandstk - 1;
         end;
   end;

procedure pushopand(element: integer);
   begin
      if topandstk = stksize then begin
         writeln('operand stack overflow');
         LLFatal; {terminate translation}
         end
      else begin
         topandstk := topandstk + 1;
         opandstk[topandstk] := element;
         end;
   end;

procedure popoptor(var result: char);
   begin
      if toptorstk = 0 then begin
         writeln('operator stack underflow');
         LLFatal; {terminate translation}
         end
      else begin
         result := optorstk[toptorstk];
         toptorstk := toptorstk - 1;
         end;
   end;

procedure pushoptor(element: char);
   begin
      if toptorstk = stksize then begin
         writeln('operator stack overflow');
         LLFatal; {terminate translation}
         end
      else begin
         toptorstk := toptorstk + 1;
         optorstk[toptorstk] := element;
         end;
   end;
```

These five function and procedure declarations should be placed in LLsup.i.

These routines reference a procedure not previously described:

LLFatal

LLFatal is a pre-defined procedure which will terminate the translation after printing an appropriate message. It is used when a catastropic translation error occurs, such as overflowing optorstk. It is also called as the default error recovery when a syntactic error is detected by skeleton.o and no user-defined recovery has been specified in the grammar. You can freely call it within your action code. It takes no arguments.

5.0 Lexical Analyzer

Once the support routines are complete, the lexical analyzer -- LLNextToken -- must be constructed. The lexical analyzer needed for translating EXPRESSIONS has been broken down into two routines, LLNextToken and nextchar. The latter is called by LLNextToken to obtain the next character from the source text and to take care of bookkeeping chores such as writing the lines of source text out to a listing file and updating a line counter.

Several pre-defined error processing routines will need to know the line number of the source text where the error occurred. This information must be kept in the pre-defined integer variable LLLineCount. Skeleton.p will initialize this counter to 0 for you before parsing begins. It is the your responsibility to update it properly through LLNextToken.

```
procedure nextchar; {assign next character from source
                      to curchar}
   begin
      if not eof then begin
         if LastWasEoln then begin
            LLLineCount := LLLineCount+1;
            LastWasEoln := false;
            end;
         if eoln then
            LastWasEoln := true;
         read(curchar);
         end {if not eof}
      else
         curchar := '@'
   end; {nextchar}


procedure LLNextToken;  {get next token from candidate}
   var
      i: integer;
   begin with LLCurtok do begin
         {curchar should become the first non-blank}
      while curchar =  ' ' do nextchar;
         {clear PrintValue field}
      for i := 1 to LLStringLength do
         PrintValue[i] := ' ';
      if curchar in ['0'..'9'] then begin
            {token is an integer}
         i := 1;
         TableIndex := LLFind('INTEGER', group);
         PrintValue[i] := curchar;
         attribute.operand :=
             ord(curchar) - ord('0');
         nextchar;
         while curchar in ['0'..'9'] do begin
            i := i+1;
            PrintValue[i] := curchar;
            attribute.operand :=
                attribute.operand*10 +
                ord(curchar) - ord('0');
            nextchar;
            end;
         end
      else if (curchar = '@') and eof then begin
         PrintValue := 'end-of-file';
         TableIndex := LLFind('@', group);
         end
      else begin
         PrintValue[1] := curchar;
         attribute.operator := curchar;
         TableIndex := LLFind(PrintValue, literal);
         nextchar;
         end;
      end; {with}
   end; {LLNextToken}
```

There are a handful of simple conventions which must be followed in constructing LLNextToken so that it communicates with the parser properly.

First, despite its appearance above LLNextToken does have a parameter. Because this routine must be referenced within skeleton.p long before the point where LLsup.i is inserted into

it, LLNextToken has a forward declaration in skeleton.p:

```
procedure LLNextToken( var LLCurTok: LLTok );
        forward;
```

All direct communication between the lexical analyzer and the parser is through the parameter LLCurTok. The type LLTok is pre-defined in skeleton.p to be:

```
LLTok = record
            PrintValue: LLStrings;
            attribute: LLattribute;
            TableIndex: integer
        end;
```

LLStrings is pre-defined to be:

```
LLStrings = packed array[1..LLStringLength] of char;
```

where LLStringLength is a pre-defined constant equal to 12. LLattribute is the user-declared type discussed in section 3.5.

LLNextToken has one major function -- to fill-in the three fields of LLCurTok. LLNextToken should assign a value to the attribute associated with the current token. The particulars of this assignment will vary with the declaration of LLattribute, the particular token encountered, and the translator being implemented. LLNextToken should assign to the PrintValue field of LLCurTok the string which you want to be printed when the built-in error-processing routines are called. For ordinary literals and groups, this is usually the characters of the candidate string. For non-printable terminals, such as an implicit end-of-statement marker as is found in FORTRAN, the string 'end-of-stmt' might be assigned to LLCurTok.PrintValue instead.

LLNextToken must also assign a value to the "TableIndex" field of the current token. Skeleton.p has an internal symbol table (not to be confused with any symbol table which you might produce for your translator) to keep information about the terminal symbols of the grammar. This table is designed to minimize parsing time. The table structure is hidden from you and is irrelevant to what you have to write in LLNextToken. All of your communication with that table will be through the pre-defined routine "LLFind":

```
function LLFind( item: LLStrings; which: LLStyle ): integer;
```

Its first argument is the literal or group name which this token corresponds to. For literals this value normally equals LLCurTok.PrintValue. For groups, LLFind should be called with the group name rather than the literal value of the token. The second argument is either the enumerated constant "group" or "literal" depending on the token type. LLFind returns the index into the symbol table where that argument can be found. If the token cannot be found, the index value 0 is returned, indicating that the token is illegal. You can process an illegal token at the lexical level if you prefer or pass the responsibility on to the parser. In any event, whether LLFind returns a positive or zero integer, this index should be assigned to LLCurTok.TableIndex.

The special case when the end of the source string is reached is handled quite simply. LLFind should be called with the first character of item equal to "@" and the remaining characters blank. "@" is a group. The value returned by LLFind should be assigned to LLCurTok.TableIndex. LLCurTok.PrintValue could be assigned the string 'end-of-file' or some other appropriate string, and LLCurTok.attribute should be left undefined.

Note that there are two user-defined variables referenced in nextchar. They must be declared in the grammar along with the other variables used for the other support routines in LLsup.i:

```
%v LastWasEoln: boolean;
   curchar: char;
```

In order for LLNextToken to work properly the first time it is called, curchar and LastWasEoln must already have a value. Hence, an action routine which assigns these two variables a value must be called before skeleton.o references LLCurTok. To do so a special user-defined procedure, ""LLInitialize" will always be executed before parsing really begins. After LLInitialize has been executed, LLNextToken will also be called automatically causing LLCurTok to become defined so that parsing can begin. LLInitialize can also be used to reset the sourcefile if it is not that standard input, or to reset or rewrite any supplemental files declared in the grammar.

Since LLInitialize will automatically be called, you should be sure to include a declaration for it in LLsup.i, even if it doesn't really do anything useful in your translator.

```
procedure LLInitialize;
   begin
     LastWasEoln := true;
     nextchar  {must be called to ensure LLNextToken works}
   end;
```

All support routines including LLNextToken are placed into LLsup.i.

If you are using a true Pascal compiler such as Berkeley's "pc" which supports separate compilation and linkage to routines written in C, (as opposed to "pi" which produces P-code, not machine code, and hence does not support separate compilation), you may want to consider writing a small C program to do the actual reading and writing and linking that with the compiled version of skeleton.p. Depending on the nature of the i/o, a C version of "nextchar" could perform significantly faster than a Pascal version. Since such a large percentage of the total time is spent reading and writing, this could dramatically affect the overall run-time of skeleton.o. Whether this particular strategy will, in fact, improve skeleton.o's performance depends heavily on the quirks of your Pascal compiler and the i/o performed in skeleton.o.

6.0 Execute generate.o and compile skeleton.p

6.1 Normal translation

At this point all pieces necessary to construct the translator are complete. Generate.o should now be executed redirecting the input from your grammar file:

```
generate.o < MyFile
```

Generate.o will print a few informational messages as it processes. In particular, it will tell you how many vocabulary symbols and productions are in your grammar. It has extensive error checking capabilities; for example, it will flag a reference to an undeclared vocabulary symbol, a second declaration of the same symbol, or the appearance of an ill-formed production. When generate.o finishes, it will return to the shell from which it was called.

At this point you should compile skeleton.p using the Pascal compiler:

```
pi skeleton.p; mv obj skeleton.o
```

All files except LLsup.i that must be included in skeleton.p will have been generated when you executed generate.o. Assuming there are no fatal error messages from the Pascal compiler, the object code should be an executable version of your translator.

The Pascal compiler may issue warnings that certain procedures which begin with "LLSkip" have not been referenced -- LLSkipToken, LLSkipNode, and LLSkipBoth. Do not be bothered by these warnings. These three procedures are pre-defined for error processing. If you use the default error recovery, they will not be referenced (which should be the case for G_EXPRESSIONS now). Later when you add error recovery information into your grammar, you will probably reference one or more of the routines, in which case the warnings will disappear.

You may receive two other warnings as well which you can safely ignore. The compiler may warn you that fields "table" and "grammar" of LLgram^ are not referenced. It is just a quirk of pi's analysis routines that it thinks these fields are never referenced. They, in fact, are referenced.

If other warning or error messages appear, they probably indicate a problem with your action code, or possibly with a grammar declaration for a variable, constant, or type. Fortunately, the Berkeley Pascal compiler pinpoints which included file it was compiling when the error was detected. You should use the following strategy to isolate errors produced during the compilation of skeleton.p:

File Where Error Occurs	Probable Problem
LLconst.i	illegal %c declaration in grammar
LLvar.i	illegal %v declaration in grammar
LLtype.i	illegal %t declaration in grammar
LLfile.i	illegal %f declaration in grammar
LLact.i	illegal action, patch, or synch code
LLsup.i	illegal support routine

Errors in the action, patch, and synchronization code will show up as problems in the file LLact.i which was produced by generate.o. (Patch and synchronization code are used in error recovery, and have a form similar to action code. For more information on them see section 7.2.) For convenience in discussing LLact.i, we will refer to any action, patch, or synchronization code as "embedded code".

LLact.i is actually the declaration of procedure LLTakeAction which structures the calls to all embedded code. LLTakeAction has the following structure:

```
procedure LLTakeAction(CaseIndex: integer);
  begin
     case CaseIndex of
        1: begin embedded-code-sequence-1 end;
        2: begin embedded-code-sequence-2 end;
           ...
        n: begin embedded-code-sequence-n end;
     end;
  end;
```

where the i-th embedded-code-sequence is a copy of the i-th embedded code sequence in the grammar beginning the count from the first production. So, for example, if the compiler reports an error in embedded-code-sequence-2 in LLact.i, then the erroneous code can be found in the 2nd embedded code sequence in your grammar.

If you discover any errors in your translator, the corrective action necessary will depend on the severity of the error. An error in the grammar will require you to modify it, reexecute generate.o, and recompile skeleton.p. However, if the grammar is correct, but one of the support routines or LLNextToken has a bug in it, then only skeleton.p needs to be recompiled after the bug is corrected.

Once skeleton.p is compiled without error, you should execute skeleton.o. The source string should come from whatever file you specified in LLNextToken. For our example this is the standard input, so we would type either:

```
skeleton.o             -- work interactively
```

or

```
skeleton.o < MyFile    -- read from file
```

Since, in this case, skeleton.o writes to the standard output, as you complete an expression and type a carriage return, the interpreter will display the expression's value on the screen. An erroneous expression will cause an error message to be displayed, and the interpreter to terminate execution. In the next section, you will learn how to specify any of several different error recovery policies instead.

6.2 Verbose mode

For convenience in debugging your translator, there is a version of the skeletal translator which includes facilities for tracing the parsing actions it takes in translating a candidate string -- skel.debug.p. These trace features are not available in skeleton.p. Samples of the "verbose" output of skel.debug.o which has the tracing features on are given in both Appendices A and B.

7.0 Error processing

A translator constructed according to the instructions presented so far will process correct input properly. However, if a string which is not in the source language is given to it, the translator will simply report the error and halt. Because of the viable prefix property of LL(1) parsers, a syntactic error is detected at the leftmost position in the source string for which there is no legitimate continuation. In most circumstances such poor error recovery (i.e., quitting) is unacceptable. Zuse has a number of other pre-defined error recovery strategies which you can specify in the grammar. These fall into two major categories: patches and synchronizations.

7.1 Patching errors

When a syntactic error is discovered, there are three easy recovery strategies other than quitting which can be tried:

skip past the current token from the source string, but continue the parse from the same place in the parse tree.

skip past the current node in the parse tree, (consider it to have been matched), but continue the parse with the same token in the source string.

skip past both the current token and the current node in the parse tree.

None of these may prove adequate, in which case the more sophisticated recovery strategy of synchronization must be used. However, local patching is often sufficient.

In order to specify a recovery strategy other than quitting, you must include what is called "patch" code in the grammar. A patch routine looks exactly like an action routine except that instead of beginning with "{a", it starts with "{p".

A patch routine can follow any terminal symbol, and applies to the symbol which precedes it. In our example, one of the alternatives for "P" can be patched quite nicely:

```
P = ( E ) {p LLSkipToken} ;
```

During normal parsing, patch routines are not executed. In this respect they differ from action routines which are always executed when it is their "turn" in the derivation. The parser simply skips over patch routines because they are not needed for normal processing.

The parser can detect a syntactic error in one of two ways. If the next symbol in the sentential form is a terminal and the current token does not match it, that is an error. Similarly, if the next symbol is a nonterminal, but no alternative production for that symbol has a selection set which includes the current token, that is also an error. Patching addresses only the first type of error, synchronization addresses both.

When the parser detects an error during the matching of a terminal in the sentential form against the current token in the source string, it checks whether there is patch code associated with that terminal symbol. Patch code is always associated with the terminal symbol it follows. In this example, patch code is associated with the right parenthesis since the "{p ...} code follows ")" in the production.

If there is no patch code, the parser quits with a fatal error message. On the other hand, if a patch is present, the parser executes the patch code. This should alter the source string by removing the current token, or treat the terminal in the sentential form as if it had been matched and advance it, or both. The parse then resumes. The patch code in the production above states that when a closing parenthesis is expected in an expression but none is found, then the current token should be skipped and the next one examined. The overall effect of this is to remove tokens until a right parenthesis is found in the source string. If the end of the source string is reached before ")", the translator will quit with a fatal error message.

LLSkipBoth, LLSkipToken, and LLSkipNode are three parameterless pre-defined procedures designed to facilitate patch recovery. They are declared in skeleton.p by:

```
procedure LLSkipBoth

procedure LLSkipToken

procedure LLSkipNode
```

LLSkipToken just removes the current token. LLSkipNode leaves

the current token in the source string unchanged, but advances the pointer to the current node in the parse tree, essentially ignoring the node in the parse tree which did not match the current token. LLSkipBoth skips past both the current token and the current node of the parse tree. All routines cause a message to be printed both to LLMessageFile and to the terminal explaining the nature of the error and the recovery taken.

LLSkipBoth can be helpful when you encounter a token which is often misplaced. For example, the following Pascal constant declarations are both incorrect:

```
const
   x : false;
   y := 3;
```

After the identifer being declared, the programmer should have written "=", but has written ":" and ":=" instead. These are likely errors, especially for a beginning programmer. Assuming that the original production for a constant declaration is:

```
ConstDcl = Ident '=' Literal
```

the translator-writer can perform special checks for these two likely errors by replacing that production with:

```
ConstDcl = Ident
         '='    {p with LLCurTok do
                   if (TableIndex = LLFind(':', literal)) or
                      (TableIndex = LLFind(':=', literal)) ther
                      LLSkipBoth
                   else
                      ... }
         Literal
```

Patches are limited because they are a highly localized recovery. Only that part of the source string immediately surrounding the current token is affected. Furthermore, parsing always resumes in the sentential form where it left off before the error was detected. Even if an entire section of the sentential form has been "contaminated" by the error, it is not possible using a patch to skip forward in the sentential form to a more appropriate point such as that corresponding to the beginning of the next statement. The need to synchronize with a point later in the sentential form motivates the recovery strategy described in the next section.

7.2 Synchronization

Synchronization is a more sophisticated recovery strategy which allows you to skip arbitrarily many tokens in the source and past arbitrarily many symbols in the sentential form before resuming the parse. It is needed when an error is so severe that local patching is inadequate and the whole "area" surrounding the error must be abandoned as non-repairable.

Synchronization recovery is specified in the grammar. As with action and patch code, synchronization information is surrounded by curly brackets, except that it begins with "{s". The productions for "Ax" can be modified from:

```
Ax =  ;

   =  {a init} E {a writeln(popopand);}
      ';' Ax ;
```

to:

```
Ax =  ;

   =  {a init;} E {s ';' => 2}
      {a writeln(popopand);} ';' Ax %any ;
```

Although only a terminal can be associated with patch code, any vocabulary symbol can be associated with a synchronization specification by placing the specification immediately after it. For our example nonterminal E is associated with the synchronization specification.

Besides including synchronization recovery, the second alternative production has been changed so that the selection set element "any" has been specified. It has been added to ensure that the second alternative production for Ax will always be selected whenever Ax must be expanded and the end of the source string has not yet been reached (the selection set of the first alternative is {@}). This is necessary because error recovery (both patch and synchronization) specified in a production only applies if that production has been previously selected. An error in the token used to select a production will cause error recovery information specified for the first vocabulary symbol on the right-hand side of that production to be ignored -- unless the selection set of that production has been augmented by "any". "any" will cause the synchronization recovery which follows the E to be applicable, even if the token used for selecting from among the alternatives for Ax is erroneous.

Synchronization information is not written in Pascal, but in a special format specifically designed to express that form of recovery. The general format is:

```
"{s"  recovery  ";"  Pascal_code  "}"
```

where each clause of the recovery specification, separated by commas, has the form:

```
token  "=>"  integer
```

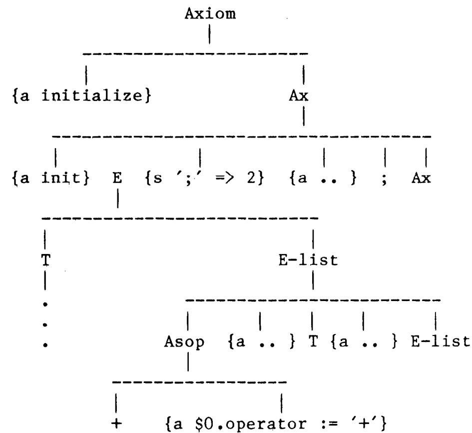
or

```
token  "=>"  "*"
```

The Pascal code which follows the semicolon will be described shortly, but first the simpler form used in G_EXPRESSIONS which has only one clause and no Pascal code will be explained.

Synchronization elements are just like patch code in that they do not affect parsing until a syntactic error is detected. The synchronization information is simply ignored until then. However, when the parser detects a syntactic error, one of several things can happen depending on the circumstances. If there is no user-defined recovery governing the parser's actions at this point, the parser treats the error as fatal and simply terminates execution. On the other hand, if the parser was attempting to match a terminal symbol against a terminal in the sentential form which has patch code associated with it, then this local recovery will be taken whether synchronization is specified or not. Under all other circumstances, synchronization recovery is activated.

To illustrate how synchronization works, suppose you were to attempt to parse the expression "3++5", which has an extra "+". When the parser looks for an operand after the first "+", it will find the operator "+" instead. The parse tree at this point, with uninteresting portions elided will be:



The parser will have successfully expanded Asop, have executed the action code following it, and be attempting to expand T when the error is detected. The current token value is "+" but the selection set for the sole alternative production for T is ['(', INTEGER]. Since T is a nonterminal, no local patch is possible. However, synchronization can be performed even though there is no synchronization specification directly following T.

T is a descendent of E-list, which in turn is a descendent of E. E is followed by a synchronization element. The synchronization policy of a nonterminal is implicitly inherited by its descendents. Any descendent nonterminal can explicitly override that synchronization policy by establishing one of its own, and any descendent terminal can override that synchronization policy through patch code, but by failing to establish a recovery of its

own, a symbol inherits its recovery policy from its parent.

The effect of this synchronization is to instruct the parser to erase that part of the parse tree hanging under the E, and to skip tokens in the source string until a semicolon is found. Failure to find a semicolon before the end of the source string is reached is fatal. However, if a semicolon is found, the parser will resynchronize the parse with the node in the parse tree corresponding to the second vocabulary symbol among the siblings of E, since it is the E where the governing synchronization is located. In this case the second vocabulary symbol is ';'. The parse then resumes as if all nonterminals to the left of that semicolon had been expanded and all terminals to its left had been matched.

The total effect of the synchronization is to skip over tokens until the end of an expression is found and to resume parsing at that point in the parse tree where the end of an expression is expected. This decision is based on the premise that if an error is discovered in an expression for which no local patch is defined, then the best policy is to simply skip past the rest of that expression and resume parsing with the expression separator.

In the more general case there will be several clauses in the synchronization specification:

```
"{s" token1 "=>" int1 "," ... tokenk "=>" intk "}"
```

Having several clauses allows the parser to resynchronize at different places depending on the sequence of tokens encountered in the source string. When synchronization recovery is initiated, the parser will skip tokens in the source string until it finds one which matches one of token1 through tokenk. When it does so, it will use that clause for synchronization, continuing the parse at the point indicated by the integer in that clause.

The advantage of being able to synchronize in different places becomes clear when we consider error recovery for a Pascal compiler. When an error occurs in a statement, the translator-writer may wish to skip to different points in the source string and parse tree depending on the circumstances. For example the variable declaration:

```
var
    t,3u = integer;
```

contains two syntax errors. The illformed token '3u' appears where an identifier is required, and '=' is used where ':' is expected. If the only error were the appearance of 3u, then one reasonable recovery would be to skip to the ':'. However, since that symbol is missing, we should skip to the ';' instead. Assuming the production in effect for the parsing of these type declarations originally were:

```
VarStmt = IdList : TypeConstruc ';' ;
```

It should become:

```
VarStmt = IdList {s ';' => 4, ':' => 2} : TypeConstruc ;
```

to implement this strategy.

Occassionally, you will not wish to synchronize on a symbol which occurs in the production in which the synchronization element appears; rather, you will want to skip past the whole right-hand side, as if the parser had completed the derivation of the entire right-hand side. To indicate this, a '*' is used in place of an integer in a clause of the synchronization element. For example, if the synchronization element of the alternative production for Ax were replaced by:

```
{s @ => *, ';' => 2}
```

then the string:

```
3/4
```

would cause the parser to recover by terminating normally even though no semicolon is present in the source. All of the children of Ax would be skipped. Since Ax is the rightmost child of Axiom, the entire parse tree would be considered generated by the parser at this point. Hence, the translation would terminate normally. Using the original synchronization element, the parser would have terminated execution with a fatal error when it was unable to find a semicolon.

An important synchronization feature is the ability to optionally include Pascal code after a semicolon in the synchronization specification. This code is occasionally necessary in order to "clean up" any data structures which would otherwise be left in an inconsistent state by the synchronization. For example, if action code on the right-hand side of a production is skipped during resynchronization, then attributes and variables which that action code would have assigned values will not have the proper values. This Pascal code will be executed after determining which clause of the synchronization specification will be used. It can assign values, clear stacks, reset counters, and perform any other housekeeping chores so that when parsing resumes, all data structures are in a consistent state. Different clean-ups are possible depending on which clause is selected. The code can examine LLCurTok to determine which clause was selected and then take the appropriate action. For example, the synchronization specification:

```
TypeDcl =
   TypeID '='
        {s ',' => 2, ':' => 3, any => * ;
        with LLCurTok do
           if TableIndex = LLFind(',' literal) then
            ... recovery appropriate for ',' ...
           else if TableIndex = LLFind(':', literal) then
            ... recovery appropriate for ':' ...
           else
            ... other recovery ...}
   TypeDenoter;
```

permits the translator to take different action when recovering from quite different situations:

```
type
   speed : integer;  {should be '=' -- assume it is}
   high,low = real;  {only one type identifier can be
                      declared per declaration -- but
                      can set up symbol table to accept
                      high and low anyhow}
   g 123 = char;     {perhaps this is an embedded blank
                      in type identifier -- don't really
                      know what to do, so skip to end of
                      declaration and "throw out" type
                      identifier g found so far}
```

There is one final embellishment on the specification of synchronization information which is often quite useful. Consider a production for a Pascal Program:

```
Program =
   Header LabelPart ConstPart TypePart VarPart FuncProcPart
   ExecStmt . ;
```

One reasonable synchronization recovery strategy is:

```
Program =
   Header
      {s PROGRAM => 1, LABEL => 2, CONST => 3, TYPE => 4,
         VAR => 5, FUNCTION => 6, PROCEDURE => 6, BEGIN => 7}
   LabelPart
      {s LABEL => 2, CONST => 3, TYPE => 4, VAR => 5,
         FUNCTION => 6, PROCEDURE => 6, BEGIN => 7}
   ConstPart
      {s CONST => 3, TYPE => 4, VAR => 5, FUNCTION => 6,
         PROCEDURE => 6, BEGIN => 7}
   TypePart
      {s TYPE => 4, VAR => 5, FUNCTION => 6, PROCEDURE => 6,
         BEGIN => 7}
   VarPart
      {s VAR => 5, FUNCTION => 6, PROCEDURE => 6, BEGIN => 7}
   FuncProcPart
      {s FUNCTION => 6, PROCEDURE => 6, BEGIN => 7}
   ExecStmt
   . ;
```

This synchronizes the program on each major block section as determined by a keyword. Although this specification is adequate, it appears highly redundant. To abbreviate the specification of recovery information common to several vocabulary symbols, Zuse allows you to write synchronization code which is global to the entire right-hand side. This code, written in the same syntax as other synchronization specifications, must appear as the first symbol on the right-hand side of the production:

```
Program =
    {s PROGRAM => 1, LABEL => 2, CONST => 3, TYPE => 4,
```

```
      VAR => 5, FUNCTION => 6, PROCEDURE => 6, BEGIN => 7}
   Header LabelPart ConstPart TypePart VarPart FuncProcPart
   ExecStmt . ;
```

The synchronization specified in this production applies to each vocabulary symbol with the following stipulation. Recall that it is illegal to attempt to synchronize to the left of the vocabulary symbol where the error is detected. Consistent with this view, when attempting to synchronize based on a global synchronization, skeleton.o will ignore a synchronization clause which would cause it to resynchronize to the left of the current vocabulary symbol. For example, the text:

```
PROGRAM test(input,output);
   LABEL 10;
   CONST
      pi : 3.14159;
   LABEL 20;
   TYPE
      speed = real;
   ...
```

contains an error in the declaration of the constant pi and erroneously has a second label declaration section. Assuming the error recovery just specified was applicable, skeleton.o would skip past the rest of the constant declarations, past the second label section, and resynchronize on the keyword TYPE. The second label declaration section would be skipped because when the error was detected, the parser would be processing ConstPart, which is the third vocabulary symbol on the right-hand side, but LABEL causes resynchronization on the second symbol.

Of course, there is not always a single synchronization strategy that is appropriate for every vocabulary symbol in a production. To accommodate this fact, you can overide global synchronization by explicitly specifying either patch or another synchronization code after any vocabulary symbol:

```
Program =
      {s PROGRAM => 1, LABEL => 2, CONST => 3, TYPE => 4,
         VAR => 5, FUNCTION => 6, PROCEDURE => 6, BEGIN => 7}
   Header LabelPart ConstPart TypePart VarPart FuncProcPart
   ExecStmt
      {s . => 8}
   . ;
```

If an error would occur in ExecStmt, then skeleton.o would skip tokens until it found a period. It would then resynchronize on the eighth vocabulary symbol. The recovery of all other vocabulary symbols would still be governed by the global synchronization specification.

Zuse Installation Instructions
Version 1.0

Arthur Pyster
May 1981

Zuse is very easy to install if you are running Unix with the Berkeley Pascal compilers (pi or pc). It was designed to be highly portable so that if you do not have these compilers, I expect you will still be able to compile Zuse without too much difficulty. Every line of Zuse which is (as best as can be determined) not standard Pascal has been marked with a comment bracketed by (* .. *). All other comments use the curly bracket delimiters { .. }. Hence, it should be quite easy to browse through the source code with a text editor and examine each non-standard feature.

Zuse consists of two Pascal programs -- generate.p and skeleton.p. Generate.p should be compiled with the resulting object code saved under the name generate.o (or whatever suits your fancy). Skeleton.p is a skeletal Pascal program which is augmented by a person writing a translator in the manner described in the Zuse Users' Manual. Hence, a user of Zuse will need access to generate.o and skeleton.p.

To create Zuse in the current directory, just type:

```
tar xv
pi [options]  generate.p
mv obj generate.o
```

where [options] are whatever compiler options you desire. You may need to qualify the tar command with the tape drive number you mount the distribution tape on. When you execute tar, a number of files other than generate.p and skeleton.p will be placed in your current directory. In particular, you will also find:

```
skel.debug.p -- debugging version of skeleton.p

RawInstall   -- pre-nroff form of this document

Install      -- post-nroff form of this document

RawUser      -- pre-nroff form of Users' Manual

UserManual   -- post-nroff form of Users' Manual

FortGrammar  -- grammar for structured FORTRAN preprocessor

FortLLsup.i  -- support routines for FORTRAN preprocessor

ExpGrammar   -- grammar for arithmetic expression evaluator

ExpLLsup.i   -- support routines for expression evaluator
```

The pre-nroff documents do not use either the -me or the -ms nroff macro libraries. They are entirely self-contained, so you can recreate either document by typing:

```
nroff RawInstall > MyInstall
nroff RawUser > MyUser
```

To create the FORTRAN preprocessor under the name Struct.o, just type:

```
generate.o < FortGrammar
cp FortLLsup.i LLsup.i
pi [options] skeleton.p
mv obj Struct.o
```

To create the expression evaluator under the name express.o, just type:

```
generate.o < ExpGrammar
cp ExpLLsup.i LLsup.i
pi [options] skeleton.p
mv obj express.o
```

A word of caution is necessary. Whenever you run generate.o, the file LLgram is created. It contains a crunched version of the grammar input to generate.o and is peculiar to the particular translator being created with Zuse. When a translator created with Zuse (such as Struct.o or express.o) is executed, it reads in LLgram in order to initialize its parser. Hence, LLgram must be in the current directory. This also means that you cannot readily have more than one translator produced by Zuse in the same directory, since each one will require its own version of LLgram. For the classroom environment for which Zuse was developed, this is no problem. However, for more varied applications, this restriction can be inconvenient. With a modest bit of surgery, this restriction is readily removed. Rename LLgram to whatever file name is convenient after running generate.o, and change the line:

```
reset(LLgram);
```

in skeleton.p to

```
reset(LLgram, YourName);
```

where YourName is the Unix file name where you moved LLgram.

```
 1. program skeleton(input,output, LLgram
 2. #include 'LLfile.i'
 3.         (* UNIX *)
 4.              );
 5.         {skeletal compiler to parse a candidate string}
 6.
 7.     { May 8, 1981
 8.       Version: 1.0
 9.       Author: Arthur Pyster
10.       Copyright (c): The Regents of the University of California
11.       Purpose: This program is a skeletal compiler which is
12.                fleshed out by the inclusion of a file supplied
13.                by the user:
14.
15.                  LLsup  -  support routines called directly
16.                            or indirectly as action routines
17.                            including LLNextToken
```

```
18.
19.                 and 5 files provided by "generate", the
20.                 parser generator:
21.
22.                     LLfile -    file dcls specified in grammar
23.
24.                     LLvar  -    var dcls required for action
25.                                 routines
26.
27.                     LLconst -   const dcls specified in
28.                                 grammar and misc. constants
29.
30.                     LLtype  -   type dcls specified in grammar
31.
32.                     LLact   -   LLTakeAction procedure which
33.                                 calls action routines as dictated
34.                                 by grammar rules
35.
36.          The file LLgram, produced by generate.o, must be read in.
37.       It contains an encoded form of the grammar, error data, and the
38.       symbol table.
39.
40.          Error messages are written to the standard output unit.}
41.
42.
43. label 1000;
44. const
45.
46. LLMaxStack = 200;  {max number of sentential form elements in parse
47.                     tree at any one time.}
48.
49. # include 'LLconst.i'
50.             (* UNIX *)
51.
52. type
53.    LLStrings = packed array[1..LLStringLength] of char;
54.
55.
56. #include 'LLtype.i'
57.             (* UNIX *)
58.
59.    LLGramEntry =
60.       record case boolean of
61.          true: (table: LLStrings);
62.          false: (grammar: integer);
63.       end;
64.
65.    LLTok =
66.       record
67.          PrintValue: LLStrings;    {the literal token value}
68.          TableIndex: integer; {index where token is in symbol table}
69.          attribute: LLattribute;
70.             {value associated with the token by
71.              LLNextToken -- can be used by an action routine}
72.       end;
73.
74.    LLStyle = (literal, nonterminal, group, action, patch);
75.                {literal: a terminal which stands for itself.}
76.                {group: a terminal which is a lexical group of LLStrings,
77.                        but syntactically just a single symbol}
78.                {action: an action routine call}
79.                {patch: an action routine to patch syntactic errors}
80.
81.    LLRight =
82.       record
83.          CaseIndex: integer;
84.          synchindex: integer;
85.          WhichChild: integer;
86.          case kind: LLStyle of
87.             action, patch: ();
88.             nonterminal: (ProdStart: integer);
89.             literal, group: (TableIndex: integer);
90.       end;
91.
92.    LLSentential =
93.       record
94.          LastChild: boolean;     {is this the rightmost child?}
95.          Top: integer;           {point to LastChild}
96.          parent: integer;        {ptr to parent of this node}
97.          attribute: LLattribute;
98.          data: LLRight;
99.       end;
100.
101. var
102. # include 'LLvar.i'
103.             (* UNIX *)
104.          {var dcls produced by parser generator}
105.    LLadvance: boolean; {advance LLSentPtr to next node?}
106.    LLStartTime: real;  {clock time at start of compilation}
107.    LLLocEOS: integer;  {location of end-of-source in SymbolTable}
108.    LLSentPtr: integer; {sentential form element currently being processed}
109.    LLLineCount: integer;           {line number of source text}
110.    LLgram: file of LLGramEntry;    {where grammar is stored}
111.    LLCurTok: LLTok;           {the current token}
112.    LLSymbolTable: array[1..LLTableSize] of
113.       record
114.          key: LLStrings;
115.          kind: LLStyle
116.       end;
117.    LLStack: array[1..LLMaxStack] of LLSentential; {stack which represents
118.                   the parse tree}
119.    LLTop: integer;  {top of stack pointer}
120.
121. procedure LLNextToken( var LLCurTok: LLTok ); forward;
122.
123.
124.
125. function LLFind( item: LLStrings; which: LLStyle ): integer;
126.         {Find item in symbol table -- return index or 0 if not found.
127.          Assumes symbol table is sorted in ascending order.}
128. label 10;
129. var
130.    low, midpoint, high: integer;
131. begin
132.    LLFind := 0; {assume failure}
133.    low := 1;
134.    high := LLTableSize;
135.    if (item >= LLSymbolTable[low].key) and
136.       (item <= LLSymbolTable[high].key) then
137.       while low < high do with LLSymbolTable[low] do begin
138.          if key = item then begin
139.             if kind = which then
140.                LLFind := low;
141.             goto 10
142.             end
143.          else begin
144.             midpoint := (high+low+1) div 2;
145.             if LLSymbolTable[midpoint].key < item then
146.                low := midpoint
147.             else if LLSymbolTable[midpoint].key > item then
148.                high := midpoint-1
149.             else if LLSymbolTable[midpoint].kind = which then begin
150.                LLFind := midpoint;
151.                goto 10
152.                end
153.             else {not in table}
154.                goto 10
155.             end;
156.          end;
157.    10:  {emergency exit}
158. end; {LLFind}
159.
160.
161. procedure LLPrtString( str: LLStrings);
162.                {print non-blank prefix of str}
163. label 10;
164. var
165.    temp: char;
166.    i: integer;
167. begin
168.    write('"');
169.    for i := 1 to LLStringLength do
170.       if str[i] = ' ' then
171.          goto 10
172.       else begin
173.          temp := str[i];
174.          write(temp);
175.          end;
176.    10:
177.    write('"');
178. end; {LLPrtString}
179.
180.
181. procedure LLHeader;    {print header message}
182. var
183.    i: integer;
184.    tempLLCurTok: LLStrings;
185. begin
186.    for i := 1 to LLStringLength do
187.       tempLLCurTok[i] := ' ';
188.    if LLCurTok.TableIndex = LLLocEOS then begin
189.       tempLLCurTok[1] := 'e'; tempLLCurTok[2] := 'o';
190.       tempLLCurTok[3] := 'f';
191.       end
192.    else if LLCurTok.PrintValue[1] in [' '..'~'] then  (* ASCII *)
193.       tempLLCurTok := LLCurTok.PrintValue;
194.    if tempLLCurTok[1] in ['!'..'~'] then (* ASCII *)
195.       LLPrtString(tempLLCurTok)
196.    else
197.       write('Unprintable token beginning with ',
198.          'ord: ', ord(tempLLCurTok[1]):3);
199. end; {LLHeader}
200.
201.
202. procedure LLSkipToken;    {remove current token}
203. begin
204.    LLadvance := false;
205.    write(LLLineCount:3, ' -- ');
206.    LLHeader;
207.    writeln( ' is skipped.');
208.    LLNextToken( LLCurTok );
209. end; {LLSkipToken}
210.
211.
212. procedure LLSkipNode; {skip over sentential form node leaving current
213.                        token as is}
214. begin
215.    write(LLLineCount:3, ' -- ');
216.    LLPrtString(LLSymbolTable[LLStack[LLSentPtr].data.TableIndex].key);
217.    write(' inserted before ');
218.    LLHeader;
219.    writeln;
220.    LLSentPtr := LLSentPtr + 1;
221. end; {LLSkipNode}
```

```
222.
223.
224. procedure LLSkipBoth; {skip over both sentential form node and current
225.                        token. used when replacement is assumed to be
226.                        correct, and attribute of replacement does not need
227.                        to be set; otherwise use LLReplace}
228. begin
229.    write(LLLineCount:3, ' -- ');
230.    LLHeader;
231.    write(' replaced by ' );
232.    LLPrtString(LLSymbolTable[LLStack[LLSentPtr].data.TableIndex].key);
233.    writeln;
234.    LLSentPtr := LLSentPtr + 1;
235.    LLNextToken(LLCurTok);
236. end;
237.
238.
239. procedure LLFatal; {to recover from syntactic error, terminate compilation}
240. begin
241.    write(LLLineCount:3, ' -- ');
242.    LLHeader;
243.    writeln(' found. Translation terminated.');
244.    goto 1000;
245. end; {LLFatal}
246.
247.
248. # include 'LLsup.i'
249.          (* UNIX *)          {supporting routines}
250.
251.
252. # include 'LLact.i'
253.              (* UNIX *)         {action code produced by parser generator}
254.
255.
256. procedure LLMain;
257. const
258.    LocOfNull = 0;       {location of null string in symbol table}
259.
260. type
261.    intset = set of 1 .. LLTableSize;
262.
263.    synchtype =
264.       record
265.          token: integer;  {index to Table entry for token}
266.          sent: integer;   {how far in LLSentential form to goto}
267.       end;
268.
269.
270.    prod =
271.       record
272.          lhs: integer; {TableIndex of lhs}
273.          rhs: integer; {index into RhsArray where rhs begins}
274.          cardrhs: integer;
275.          select: intset;
276.          cardsel: integer;
277.       end;
278.
279. var
280.    ThisRhs: integer;  {index into RhsArray}
281.    RhsArray: array[1..LLRhsSize] of LLRight;  {rhs elements of productions}
282.    synchdata: array[0..LLSynchSize] of synchtype;
283.    axiom: integer;  {pointer to first production whose lhs is the axiom}
284.    productions: array[1 .. LLProdSize] of prod;
285.
286. procedure readgram;  {read grammar from disk}
287. var
288.    i: integer;
289.
290.
291. procedure BuildRight(whichprod: integer);  {establish contents of rhs}
292. var
293.    ChildCount: integer;    {which # child in rhs is this?}
294.    i: integer;
295.    temp: integer;
296.
297. begin with productions[whichprod] do begin
298.    rhs := ThisRhs+1;
299.    ChildCount := 0;
300.    for i := ThisRhs+1 to ThisRhs+cardrhs do
301.       if i <= LLRhsSize then with RhsArray[i] do begin
302.          ThisRhs := ThisRhs+1;
303.          get(LLgram);
304.          temp := LLgram^.grammar;  {the type of symbol}
305.          get(LLgram);         {info for that particular symbol type}
306.          case chr(temp) of
307.             'l': begin
308.                ChildCount := ChildCount+1;
309.                WhichChild := ChildCount;
310.                kind := literal;
311.                TableIndex := LLgram^.grammar;
312.                get(LLgram);
313.                CaseIndex := LLgram^.grammar;
314.                get(LLgram);
315.                synchindex := LLgram^.grammar;
316.                end;
317.             'a': begin
318.                kind := action;
319.                CaseIndex := LLgram^.grammar;
320.                end;
321.             'n': begin
322.                ChildCount := ChildCount+1;
323.                WhichChild := ChildCount;
324.                kind := nonterminal;
325.                ProdStart := LLgram^.grammar;
326.                get(LLgram);
327.                CaseIndex := LLgram^.grammar;
328.                get(LLgram);
329.                synchindex := LLgram^.grammar;
330.                end;
331.             'g': begin
332.                ChildCount := ChildCount+1;
333.                WhichChild := ChildCount;
334.                kind := group;
335.                TableIndex := LLgram^.grammar;
336.                get(LLgram);
337.                CaseIndex := LLgram^.grammar;
338.                get(LLgram);
339.                synchindex := LLgram^.grammar;
340.                end;
341.             'p': begin
342.                kind := patch;
343.                CaseIndex := LLgram^.grammar;
344.                end;
345.             end; {case}
346.             end {with RhsArray}
347.          else begin {grammar in LLgram is screwed up}
348.             writeln( 'Catastrophic error -- The grammar used to ');
349.             writeln( 'generate this compiler probably had an error.');
350.             writeln( 'Check to make sure that "generate.o" did not ');
351.             writeln( 'produce any error messages when it processed ');
352.             writeln( 'your grammar.');
353.             goto 1000
354.             end;
355.
356.    end; {with productions}
357. end; {BuildRight}
358.
359.
360. procedure BuildSelect(whichprod: integer);  {build selection set}
361. var
362.    i: integer;    {loop counter}
363.    TableIndex: integer;  {where in Table can element be found?}
364. begin with productions[whichprod] do begin
365.    select := [];
366.    for i := 1 to cardsel do begin
367.       get(LLgram);
368.       TableIndex := LLgram^.grammar;
369.       select := select + [TableIndex];
370.       end; {for i}
371.    end; {with gram}
372. end; {BuildSelect}
373.
374.
375. begin {readgram}
376.     {read in symbol table}
377.    reset(LLgram);
378.    if LLTableSize > 0 then begin
379.       LLSymbolTable[1].key := LLgram^.table;
380.       get(LLgram);
381.       if LLgram^.table[1] = 'g' then
382.          LLSymbolTable[1].kind := group
383.       else
384.          LLSymbolTable[1].kind := literal;
385.       end;
386.    for i := 2 to LLTableSize do begin
387.       get(LLgram);
388.       LLSymbolTable[i].key  := LLgram^.table;
389.       get(LLgram);
390.       if LLgram^.table[1] = 'g' then
391.          LLSymbolTable[i].kind := group
392.       else
393.          LLSymbolTable[i].kind := literal;
394.       end; {for i}
395.
396.     {read in grammar}
397.    ThisRhs := 0;
398.    get(LLgram);
399.    axiom := LLgram^.grammar;
400.    for i := 1 to LLProdSize do with productions[i] do begin
401.       get(LLgram); lhs := LLgram^.grammar;
402.       get(LLgram); cardrhs := LLgram^.grammar;
403.       BuildRight(i);
404.       get(LLgram); cardsel := LLgram^.grammar;
405.       BuildSelect(i);
406.       end; {with}
407.               {now read in synchronization info}
408.    for i := 1 to LLSynchSize do begin
409.       get(LLgram);
410.       synchdata[i].token := LLgram^.grammar; {LLSymbolTable location}
411.       get(LLgram);
412.       synchdata[i].sent := LLgram^.grammar;  {where do I skip to?}
413.       end; { for i}
414. end; {readgram}
415.
416.
417. procedure parse;   {parse the candidate}
418. var
419.    temp: LLStrings;
420.    LocOfAny: integer;      {location of "any" in LLSymbolTable}
421.    i: integer;             {loop counter}
422.
423. procedure erase;
424.          {has rhs of prod has been matched? if so then erase rhs}
425. label 10;
```

```
426. begin {only erase if at farthest point to the right in a production}
427.    if LLStack[LLSentPtr].LastChild then begin
428.       while LLStack[LLSentPtr].LastChild do begin {erase rhs}
429.          LLSentPtr := LLStack[LLSentPtr].parent;
430.          if LLSentPtr = 0 then begin {stack is empty}
431.             LLTop := 0;
432.             LLadvance := false; {don't try to advance beyond axiom}
433.             goto 10;
434.             end;
435.          end;
436.       LLTop := LLStack[LLSentPtr].Top; {set LLTop to be the LastChild
437.                    of current rhs}
438.       10:
439.       end;
440. end; {erase}
441.
442.
443. procedure testsynch; forward;
444.
445. procedure expand; {expand nonterminal in sentential form}
446. var
447.    i: integer;              {loop counter}
448.    where: integer;          {production being examined}
449.    OldTop: integer;         {top of stack ptr before expansion}
450.
451. function match(sentindex: integer): integer;
452.          {does a production whose lhs is sentindex and whose
453.           selection set includes token exist?
454.           If so, return index to that production as value of match;
455.           otherwise, set match to 0.}
456.
457. label 10;
458. var
459.    i: integer;     {loop counter}
460. begin
461.    match := 0;     {assume failure and reset if successful}
462.    for i := sentindex to LLProdSize do with productions[i] do
463.       if lhs = sentindex  then {production has proper lhs}
464.          if (LLCurTok.TableIndex in select) or (LocOfAny in select) then
465.             begin
466.             match := i;
467.             goto 10;
468.             end {if LLCurTok}
469.          else
470.       else
471.          goto 10;
472.    10:  {emergency exit point}
473. end; {match}
474.
475.
476. begin {expand}
477.    where := match(LLStack[LLSentPtr].data.ProdStart);
478.    if where <> 0 then with productions[where] do
479.              {rhs of new production will be placed in list}
480.       if cardrhs > 0 then begin
481.          LLadvance := false;
482.          OldTop := LLTop;
483.          if LLTop + cardrhs > LLMaxStack then begin {overflow}
484.             writeln('Internal stack overflow. Recompile skeleton after',
485.                     ' increasing constant LLMaxStack');
486.             LLFatal;
487.             end;
488.          for i := 1 to cardrhs do begin
489.             LLTop := LLTop + 1;
490.             with LLStack[LLTop] do begin
491.                parent := LLSentPtr;
492.                   {put data into children from the selected production}
493.                data := RhsArray[rhs+i-1];
494.                LastChild := false;
495.                case data.kind of
496.                   action, patch, literal, group:;
497.                   nonterminal:
498.                      Top := OldTop + cardrhs;
499.                end; {case}
500.                end; {with LLStack[LLTop]}
501.             end; {for i}
502.                   {mark rightmost child as the last}
503.          LLStack[LLTop].LastChild := true;
504.                   {move LLSentPtr to the first new child}
505.          LLSentPtr := OldTop + 1;
506.          end {if}
507.       else
508.    else
509.       testsynch;
510. end; {expand}
511.
512.
513. procedure testsynch;
514.
515.
516. procedure synchronize;
517.       {synchronize token and LLSentential form to recover from syntactic
       error}
518. label 10;
519. var
520.    OldCurTokIndex: integer;
521.    i: integer;
522.    temp: LLStrings;
523.    LocOfAny: integer;
524. begin
525.    write(LLLineCount:3, ' -- ');
526.    LLHeader;
527.    writeln(' unexpectedly encountered.');
528.    OldCurTokIndex := LLCurTok.TableIndex;
529.    for i := 1 to LLStringLength do temp[i] := ' ';
530.    temp[1] := 'a'; temp[2] := 'n'; temp[3] := 'y';
531.    LocOfAny := LLFind(temp, group);
532.    repeat
533.       i := LLStack[LLSentPtr].data.synchindex;
534.       while synchdata[i].sent <> 0 do begin with synchdata[i] do
535.          if ((LLCurTok.TableIndex = token) or (token = LocOfAny)) and
536.             (LLStack[LLSentPtr].data.WhichChild <= sent)  then begin
537.             if LLCurTok.TableIndex = LLLocEOS then begin
538.                writeln('       Translation terminated.');
539.                goto 1000;
540.                end;
541.             if LLCurTok.TableIndex <>  OldCurTokIndex then begin
542.                write('       Skipping to ');
543.                LLPrtString(LLCurTok.PrintValue);
544.                writeln(' in line ', LLLineCount:3);
545.                end;
546.             if LLStack[LLSentPtr].data.CaseIndex <> 0 then begin
547.                      {execute code after ";"}
548.                LLTakeAction(LLStack[LLSentPtr].data.CaseIndex);
549.                end;
550.             if sent <> maxint then begin  {skip to correct symbol in rhs}
551.                while LLStack[LLSentPtr].data.WhichChild <> sent do
552.                   LLSentPtr := LLSentPtr + 1;
553.                LLadvance := false;
554.                end {if sent}
555.             else begin  {skip to rightmost node and signal reduction}
556.                while not LLStack[LLSentPtr].LastChild do
557.                   LLSentPtr := LLSentPtr + 1;
558.                end;
559.             goto 10;
560.             end; {then}
561.          i := i+1;
562.          end; {while}
563.       if LLCurTok.TableIndex <> LLLocEOS then
564.       LLNextToken(LLCurTok)
565.       else begin
566.          writeln(' Translation terminated.');
567.          goto 1000;
568.          end
569.    until false;
570.    10:    {exit point for loop}
571. end; {synchronize}
572.
573.
574. begin {testsynch}
575.    while LLStack[LLSentPtr].data.synchindex = 0 do {no synch info there}
576.       if LLStack[LLSentPtr].parent <> 0 then  {there really is a parent}
577.       LLSentPtr := LLStack[LLSentPtr].parent
578.       else
579.          LLFatal;
580.    synchronize;
581. end; {testsynch}
582.
583.
584. begin {parse}
585.    LLSentPtr := 1;    {initialize sentform to be axiom}
586.    LLTop := 1;
587.    with LLStack[LLSentPtr], data do begin
588.       ProdStart := axiom;
589.       LastChild := true;
590.       kind  := nonterminal;
591.       synchindex := 0;
592.       end;
593.
594.             {find location of 'any' in LLSymbolTable}
595.    for i := 4 to LLStringLength do
596.       temp[i] := ' ';
597.    temp[1] := 'a'; temp[2] := 'n'; temp[3] := 'y';
598.    LocOfAny := LLFind(temp, group);
599.
600.             {find location of end-of-source represented by
601.              "@" in LLSymbolTable}
602.    temp[1] := '@'; temp[2] := ' '; temp[3] := ' ';
603.    LLLocEOS := LLFind(temp, group);
604.
605.    LLLineCount := 0;
606.    LLInitialize;
607.    LLNextToken(LLCurTok);  {call lexical analyzer the first time}
608.    while LLTop <> 0 do begin {derivation isn't finished}
609.       LLadvance := true;   {presume LLSentPtr advanced after iteration}
610.       with LLStack[LLSentPtr], data do begin
611.          case kind of
612.             group, literal: begin
613.                if TableIndex = LLCurTok.TableIndex then begin
614.                   attribute := LLCurTok.attribute;
615.                   LLNextToken(LLCurTok);
616.                   end {else if}
617.                else if TableIndex = LocOfNull then begin
618.                   {do nothing} end
619.                else if not LLStack[LLSentPtr].LastChild then
620.                   if LLStack[LLSentPtr + 1].data.kind = patch then
621.                      LLTakeAction(LLStack[LLSentPtr +1].data.CaseIndex)
622.                   else testsynch
623.                else testsynch;
624.                end; {group, literal}
625.             nonterminal: expand;
626.             action: LLTakeAction(LLStack[LLSentPtr].data.CaseIndex);
627.             patch: ;
628.             end; {case}
```

```
629.          end; {with}
630.       if LLadvance then begin
631.                {Finished with current LLStack[LLSentPtr].
632.                 Move on to next node in tree}
633.          erase;
634.          LLSentPtr := LLSentPtr + 1;
635.          end;
636.       end; {while}
637.    if LLCurTok.TableIndex <> LLLocEOS then LLFatal;
638.          {only matched against part of candidate, which is not a sentence.
639.           terminate parsing action.}
640. end; {parse}
641.
642.
643. begin  {LLMain}
644.    readgram;   {get the grammar from the user.}
645.    parse;
646. end; {LLMain}
647.
648.
649. begin {main program}
650.    LLStartTime := clock; (* UNIX *)
651.    LLMain;
652. 1000:
653.    writeln;
654.    writeln('**** End of translation. ', (clock-LLStartTime)/1000.0:5:1,
655.             ' seconds CPU time ****'); (* UNIX *)
656. end.
```

```
  1. program generate (input, output, LLgram, ConstFile, VarFile,
  2.                   ActFile, TypeFile, FileFile, LLselect);
  3.
  4.         {  May 8, 1981
  5.            Version: 1.0
  6.            Author: Arthur Pyster
  7.            Copyright (c): The Regents of the University of California
  8.            Purpose: Accept the specification of a context-free translation
  9.                     grammar from standard input file and output either
 10.                     6 or 7 files:
 11.
 12.                        LLgram   - a processed form of the grammar which
 13.                                   will be read by SKELETON
 14.                        LLvar    - Pascal var declarations which will be
 15.                                   included in SKELETON.
 16.                        LLconst  - Pascal constant declarations which will
 17.                                   be included in SKELETON.
 18.                        LLtype   - Pascal type declarations which will be
 19.                                   included in SKELETON.
 20.                        LLfile   - Pascal file declarations for program
 21.                                   statement which will be included in
 22.                                   SKELETON.
 23.                        LLact    - The TAKEACT procedure, included in
 24.                                   SKELETON, which calls action
 25.                                   code as dictated by the grammar.
 26.                        LLselect - On request (%s in grammar), formatted
 27.                                   selection sets.  }
 28.
 29. label 1000; {emergency exit for unrecoverable error}
 30. const
 31.    StringSize = 12;    {max length of vocabulary symbols}
 32.    ProdSize = 160;     {max number of productions}
 33.    SynchSize = 40;     {max number of synchronization elements}
 34.    RhsSize = 330;      {max number of rhs elements of productions}
 35.    TableSize = 135;    {max number of nonterminals, literals, and groups}
 36.    EndOfSource = '@';  {end of candidate marker}
 37.
 38. type
 39.    setofchar = set of char;
 40.          {indicates current knowledge of whether production or symbol
 41.           is nullable}
 42.    nulltypes = (notsure, never, null);
 43.    intset = set of 1..TableSize;
 44.    strings = packed array[1..StringSize] of char;
 45.    nonneg = 0..maxint;     {non-negative integers}
 46.    positive = 1..maxint;  {positive integers}
 47.    synchtype =  {synchronization info}
 48.       packed record
 49.          TableIndex: integer; {which literal or group?}
 50.          sent: nonneg;        {where do we go in sentential form}
 51.       end;
 52.    style = (action, group, literal, nonterminal, patch);
 53.             {literal: a terminal which stands for itself.
 54.              group:   a terminal which is a lexical group of strings,
 55.                       but syntactically just a single symbol.
 56.              action:  action code
 57.              patch:   action code called to repair a syntax error}
 58.
 59.    RhsElement = {info about symbol of grammar}
 60.       record
 61.          SynchIndex: nonneg;  {for vocabulary symbols points to place
 62.                   in syncharray where synch data about this symbol is
 63.                   located; not used for others}
 64.          CaseSelect: nonneg;
 65.                   {for vocabulary symbols this is the synch code LLact
 66.                    index; for action and patch this is the LLact index}
 67.          case kind: style of
 68.             nonterminal, literal, group: (TableIndex: nonneg;
 69.                   WhichVocabSymbol: nonneg);
 70.                   {index to symbol table entry and relative count of
 71.                    vocab symbols on rhs of production}
 72.                   action, patch: ()
 73.       end;
 74.
 75.    prod = {a production}
 76.       record
 77.          line: integer;  {line number where production begins}
 78.          lhs: integer;   {index to SymbolTable entry for lhs of prod}
 79.          rhs: nonneg; {index into RhsArray}
 80.          CardRhs: 0..RhsSize;  {number of rhs elements}
 81.          select: intset;   {selection set elements}
 82.          resolve: intset;    {elements to be forced into selection
 83.                               set by user directive}
 84.          CardSel: 0..TableSize; {number of selection set elements}
 85.          nullable: nulltypes;  {is production nullable?}
 86.       end;
 87.
 88.    symbol =
 89.       record
 90.          value: strings;
 91.          ProdStart: nonneg;
 92.          case kind: style of
 93.             nonterminal: (nullable: nulltypes);
 94.             literal, group, action, patch: ()
 95.       end;
 96.
 97.    GramEntry =
 98.       record case boolean of
 99.          true: (table: strings);
100.          false: (grammar: integer);
101.       end;
102.
103. var
104.    PrintSelect: boolean;  {print selection sets of productions?}
105.    ErrorFree: boolean; {any error in grammar?}
106.    axiom: nonneg;
107.    SymbolTable: packed array[1..TableSize] of symbol;
108.    RhsArray: packed array[1..RhsSize] of RhsElement;
109.    CardSymbol: nonneg; {number of entries in SymbolTable}
110.    CardSynch: nonneg; {current place in SynchData array being addressed}
111.    production: packed array[1..ProdSize] of prod;
112.    ThisRhs: nonneg;    {number of rhs elements in all prods}
113.    CardProd: nonneg;  {number of productions}
114.    AllBlanks: strings; {StringSize blanks}
115.    eolninput: boolean; {eoln(input) ?}
116.    i: integer;   {loop counter}
117.    spacers: setofchar;  {blank and tab chars}
118.    linecount: nonneg;  {how many lines of grammar have been read}
119.    LLselect: file of char; {where selection sets are
120.          printed on request (%s in grammar)}
121.    LLgram: file of GramEntry;     {where compact form of grammar is kept}
122.    FileFile: file of char;        {where file dcls are stored}
123.    ActFile: file of char;         {where action routines are stored}
124.    ConstFile: file of char;       {where grammar defined constants for
125.                                    inclusion in SKELETON go}
126.    VarFile: file of char;         {where grammar defined vars for
127.                                    inclusion in SKELETON go}
128.    TypeFile: file of char;        {where grammar defined types for
129.                                    inclusion in SKELETON go}
130.    nextact: integer;              {case number of next action routine}
131.    SynchData: packed array[1..SynchSize] of synchtype;
132.             {where synchronization data is stored until
133.              written to LLgram at the end of grammar processing}
134.
135.
136. procedure SortTable;  {sort the symbol table into ascending order by
137.                          value field.}
138. label 10;
139. var
140.    i,j: integer;
141.    ChangeMade: boolean;
142.    temp: symbol;
143. begin
144.    for i := 1 to CardSymbol-1 do begin
145.       ChangeMade := false;
146.       for j :=  1 to CardSymbol-i do
147.          if SymbolTable[j].value > SymbolTable[j+1].value then begin
148.                   {exchange}
149.             ChangeMade := true;
150.             temp := SymbolTable[j];
151.             SymbolTable[j] := SymbolTable[j+1];
152.             SymbolTable[j+1] := temp;
153.             if axiom = j then axiom := j+1
154.             else if axiom = j+1 then axiom := j;
155.             end;
156.       if not ChangeMade then goto 10
157.       end;
158.    10:
159. end;
160.
161. function OrderedFind( newvalue: strings): nonneg;
162.          {Find location of newvalue in SymbolTable.
163.           Return index if found; otherwise, return 0.
164.           Assumes table is sorted in increasing order.}
165. label 10;
166. var
167.    low, midpoint, high: integer;
168. begin
169.    OrderedFind := 0;  {presume failure}
170.    low := 1;
171.    high := CardSymbol;
172.    if (newvalue <= SymbolTable[high].value) and
```

```
173.       (newvalue >= SymbolTable[low].value) then
174.       while low < high do
175.          if SymbolTable[low].value = newvalue then begin
176.             OrderedFind := low;
177.             goto 10
178.             end
179.          else begin
180.             midpoint := (low+high+1) div 2;
181.             with SymbolTable[midpoint] do
182.                if value > newvalue then
183.                   high := midpoint-1
184.                else if value < newvalue then
185.                   low := midpoint
186.                else begin {value = newvalue}
187.                   OrderedFind := midpoint;
188.                   goto 10
189.                   end
190.             end;
191.     10:
192. end;
193.
194. function find( newvalue: strings): nonneg;
195.             {find location of newvalue in SymbolTable.
196.              Return index if found; otherwise, return 0}
197. label 10;
198. var
199.    i: positive;
200. begin
201.    find := 0;    {presume it is not found}
202.    for i := 1 to CardSymbol do
203.       if SymbolTable[i].value = newvalue then begin
204.          find := i;
205.          goto 10;
206.          end;
207.    10:
208. end; {find}
209.
210.
211. function PrintString( s: strings ): char;
212.          {print s to LLselect}
213. label 10;
214. var
215.    i: positive;
216. begin
217.    PrintString := ' ';
218.    for i := 1 to StringSize do begin
219.       if s[i] = ' ' then
220.          goto 10;
221.       write(LLselect, s[i]);
222.       end;
223.    10:
224. end; {PrintString}
225.
226. procedure insert( var newvalue: strings;
227.                       newkind: style);
228.            {insert entry into SymbolTable}
229. begin
230.    CardSymbol := CardSymbol+1;
231.    if CardSymbol <= TableSize then
232.       with SymbolTable[CardSymbol] do begin
233.          value := newvalue;
234.          ProdStart := 0;
235.          kind := newkind;
236.          end {with SymbolTable}
237.    else begin
238.       writeln('symbol table overflow -- recompile "generate"');
239.       writeln('after increasing the constant "TableSize"');
240.       goto 1000;
241.       end;
242. end; {insert}
243.
244.
245. procedure DoGrammar; {get the grammar from the user}
246. const
247.    LongStringSize = 120;
248. var
249.    CurLine: array[1..LongStringSize] of char;
250.    NextChar: char;   {next character in line from input}
251.    ch: char;      {current character of line from input}
252.    i: nonneg;    {loop counter}
253.    LineLength: integer;
254.    LineMarker: integer;
255.
256.
257. procedure readchar;
258. var
259.    tmp: char;
260. begin
261.    if LineMarker >= LineLength then begin
262.       if eof(input) then begin
263.          writeln(linecount: 3, ' --  unexpected end of input');
264.          goto 1000;  {emergency exit}
265.          end;
266.       LineLength := 0;
267.       while not eoln(input) do begin  {read in line}
268.          LineLength := LineLength+1;
269.          read(tmp);
270.                {only fill up through LongStringSize-2 chars since
271.                 last two slots are filled with blanks later}
272.          if LineLength <= LongStringSize - 2 then
273.             CurLine[LineLength] := tmp
274.          else if LineLength = LongStringSize - 1 then
275.             writeln(linecount: 3, ' -- line longer than ',
276.                        (LongStringSize-2): 3, ' characters. Rest is ignored.');
277.          end; {while}
278.       readln;
279.       if LineLength > LongStringSize-2 then {reset it}
280.          LineLength := LongStringSize-1
281.       else
282.          LineLength := LineLength+1;    {count eoln in line}
283.       CurLine[LineLength] := ' ';    {make eoln a blank}
284.       CurLine[LineLength+1] := ' ';  {make NextChar of last char a blank}
285.       LineMarker := 0;
286.       linecount := linecount+1;
287.       end;
288.    LineMarker := LineMarker+1;
289.    ch := CurLine[LineMarker];
290.    NextChar := CurLine[LineMarker+1];
291.    if LineMarker=LineLength then
292.       eolninput := true
293.    else
294.       eolninput := false;
295. end; {readchar}
296.
297.
298.
299. function printable( s: strings ): char;
300.          {print s if printable; otherwise print ord(s)}
301. label 10;
302. var
303.    i: positive;
304. begin
305.    printable := ' ';
306.    write(linecount:3, ' -- "');
307.    for i := 1 to StringSize do
308.       if s[i] = ' ' then
309.          goto 10
310.       else if s[i] in ['!'..'~'] then write(s[i]) (* ASCII *)
311.       else begin
312.          write('ord: ', ord(s[i]): 3);
313.          goto 10
314.          end;
315.    10: write('"');
316. end; {printable}
317.
318.
319. procedure fillid(var id: strings; {build an id from input}
320.                    block: setofchar);
321. var
322.    i: 0..maxint; {number of chars in id}
323.    quoted: boolean;   {is the id surrounded by quotes?}
324. begin
325.    id := AllBlanks;
326.    i := 0;
327.    if ch = '''' then begin {id is surrounded by quotes -- just ignore them}
328.       readchar;  {skip past beginning quote}
329.       quoted := true
330.       end
331.    else
332.       quoted := false;
333.    while ((not quoted) and (not (ch in block))) or
334.          (quoted and (ch <> '''')) do begin
335.       i := i + 1;
336.       if i = StringSize+1 then begin
337.          writeln(linecount:3, ' -- symbol beginning with ',
338.                id, ' exceeds maximum permissible length of ');
339.          writeln('       ', StringSize:2, '. Only first ', StringSize:2,
340.                ' characters examined.' );
341.          end
342.       else if i <= StringSize then
343.          id[i] := ch;
344.       readchar;
345.       end;
346.    if quoted then  {skip past closing quote}
347.       readchar;
348. end; {fillid}
349.
350.
351. procedure skipspace;    {skip comments, spaces and tabs}
352. begin
353.    repeat
354.       while ch in spacers do readchar;
355.       while  (ch = '(') and (NextChar = '*') do begin {skip comments}
356.       readchar; readchar;  {skip past '(' and '*'}
357.       while (ch <> '*') or (NextChar <> ')') do readchar;
358.       readchar; readchar;
359.       end
360.    until not (ch in spacers);
361. end; {skipspace}
362.
363.
364. procedure declarations;  {process the declarations}
365.       {procedure reads grammar defn from first char up through
366.        and including %%. ch will equal char following %% when
367.        procedure terminates.}
368. var
369.    temp: strings;
370.    selector: char;    {second character of dcl --
371.                         specifies type of declaration}
372.    next: strings;     {next token in line}
373.
374.
```

```
375. function firstdcl( candidate: strings ): boolean;
376.         {has the candidate been declared yet?}
377. var
378.    i: integer;  {loop counter}
379.    continue: boolean;
380.    where: nonneg;
381. begin
382.    firstdcl := true;
383.    where := find(candidate);
384.    if where > 0 then begin
385.       firstdcl := false;
386.       write( linecount: 3, ' -- ');
387.       case SymbolTable[where].kind of
388.          nonterminal: write( 'nonterminal ');
389.          literal: write( 'literal ');
390.          group: write( 'group ');
391.          end;
392.       write('"');
393.       i := 1;
394.       continue := true;
395.       while continue do begin
396.          if i = StringSize then
397.             continue := false
398.          else if candidate[i] = ' ' then
399.             continue := false
400.          else
401.             write(candidate[i]);
402.          i := i+1;
403.          end; {while}
404.       writeln( '" already declared. Second dcl ignored.');
405.       ErrorFree := false;  {fatal error}
406.       end; {if}
407. end; {firstdcl}
408.
409.
410. begin {declarations}
411.    CardSymbol := 0;  {no symbols declared yet}
412.    spacers := [' ', chr(9)];     {blank and tab}  (* ASCII *)
413.    axiom := 0;   {initially axiom is null}
414.    temp := AllBlanks;
415.    temp[1] := EndOfSource;       {insert end of candidate symbol}
416.    insert(temp, group);
417.          {insert "any"}
418.    temp[1] := 'a'; temp[2] := 'n'; temp[3] := 'y';
419.    insert(temp, group);
420.    rewrite(VarFile, 'LLvar.i');              (* UNIX *)
421.    rewrite(TypeFile, 'LLtype.i');            (* UNIX *)
422.    rewrite(FileFile, 'LLfile.i');            (* UNIX *)
423.    readchar;
424.    skipspace;   {find first non-spacer in dcl section}
425.    if ch <> '%' then begin
426.       ErrorFree := false;
427.       writeln(linecount: 3, ' -- should begin dcl with "%".',
428.            ' Skipping until "%" is found');
429.       while ch <> '%' do readchar;
430.       end; {if ch}
431.    PrintSelect := false;  {presume no selection set printout}
432.    repeat         {process one declaration at a time until end of dcls}
433.       readchar;       {read selector}
434.       selector := ch;
435.       if selector <> '%' then begin  {not end of dcl section}
436.          if selector in ['A'..'Z'] then     {change u.c. to l.c.}
437.             selector := chr(ord(selector)+32); (* ASCII *)
438.          if selector = 's' then begin {print selection sets}
439.             PrintSelect := true;
440.             rewrite(LLselect); {prepare to write formatted grammar}
441.             repeat
442.                readchar
443.             until ch = '%';
444.             end
445.          else if selector in ['a', 'g', 'l', 'n'] then  begin
446.                {not for llvar.i, llconst.i or lltype.i}
447.             readchar;  {skip past selector}
448.             skipspace;
449.             repeat
450.                fillid(next, spacers);
451.                if firstdcl(next) then
452.                   case selector of
453.                      'n': begin
454.                         insert(next, nonterminal);
455.                         SymbolTable[find(next)].nullable := notsure;
456.                         end;
457.                      'a':
458.                         if axiom = 0 then begin {dcl axiom for first time}
459.                            insert(next, nonterminal);
460.                            SymbolTable[find(next)].nullable := notsure;
461.                            axiom := find(next);
462.                            end
463.                         else  {axiom being redeclared}
464.                            writeln( linecount:3,
465.                               ' -- axiom declared for second ',
466.                               'time. Second dcl ignored.' );
467.                      'l': insert(next, literal);
468.                      'g': insert(next, group);
469.                      end; {case}
470.                skipspace
471.             until ch = '%'
472.             end
473.          else if selector in ['f', 'v', 't', 'c'] then  begin
474.                {it is a dcl for inclusion in LLvar, LLtype, LLfile,
475.                 or LLconst}
476.             readchar;   {skip past selector}
477.             repeat
478.                case selector of
479.                   'c': write(ConstFile, ch);
480.                   't': write(TypeFile, ch);
481.                   'v': write(VarFile, ch);
482.                   'f': write(FileFile, ch);
483.                   end; {case}
484.                if eolninput then
485.                   case selector of
486.                      'c': writeln(ConstFile);
487.                      't': writeln(TypeFile);
488.                      'v': writeln(VarFile);
489.                      'f': writeln(FileFile);
490.                      end; {case}
491.                readchar;
492.             until ch = '%';
493.             end
494.          else begin {bad selector}
495.             ErrorFree := false;
496.             writeln(linecount:3 , ' -- bad selector "', selector,
497.                   '". Skipping to next "%"');
498.             while ch <> '%' do readchar;
499.             end;
500.          end
501.    until selector = '%';
502.    readchar;  {skip past '%'}
503.    if axiom = 0 then begin
504.       ErrorFree := false;
505.       writeln('You forgot to declare an axiom for your grammar.');
506.       end;
507. end; {declarations}
508.
509.
510. procedure DoProductions;
511.       {Procedure assumes that ch is first char after %%. It reads
512.        up through next %%. ch will be second %}
513.
514. procedure DoLeftHandSide;   {get lhs of production}
515. var
516.    where: nonneg;
517.    temp: strings; {string value of lhs}
518. begin with production[CardProd] do begin
519.    skipspace;     {skip initial blanks}
520.    line := linecount;
521.    lhs := 0;      {always assign lhs a value, even if it is not legal}
522.    if ch <> '=' then begin
523.       fillid(temp, spacers+['=']);
524.       where := OrderedFind(temp);
525.       if where > 0 then begin
526.          if SymbolTable[where].kind <> nonterminal then begin
527.             ErrorFree := false;
528.             writeln(printable(temp), ' should be a nonterminal')
529.             end
530.          else if SymbolTable[where].ProdStart = 0 then
531.                {first alternative for lhs}
532.             SymbolTable[where].ProdStart := CardProd
533.          else begin
534.             ErrorFree := false;
535.             writeln(printable(temp), ' should not have second',
536.                   ' set of productions.');
537.             end;
538.          lhs := where
539.          end
540.       else begin
541.          ErrorFree := false;
542.          writeln(printable(temp), ' is undeclared.');
543.          end;
544.       skipspace;
545.       end {if ch}
546.    else if CardProd > 1 then {not the first production}
547.          {use lhs of previous production}
548.       lhs := production[CardProd-1].lhs
549.    else begin
550.       ErrorFree := false;
551.       writeln(linecount:3, ' -- First production should ',
552.             'have explicit right-hand side.');
553.       end;
554.    if ch <> '=' then begin     {illegal form of production}
555.       temp := AllBlanks;
556.       temp[1] := ch;
557.       ErrorFree := false;
558.       writeln(printable(temp), ' found, but "=" expected after ',
559.             'left-hand side of production')
560.       end
561.    else
562.       readchar;   {skip past =}
563.    skipspace;
564.    end; {with}
565. end; {DoLeftHandSide}
566.
567.
568. procedure DoRightHandSide; {get the rhs of the production}
569. var
570.    TotalVocab: nonneg;   {how many rhs nonterms and terms seen so far?}
571.    name: strings;
572.    ProdSynchIndex: integer;
573.    ProdCaseSelect: integer;
574.
575. procedure CopyAction;   {copy action, patch, or synch code to LLact}
576.         {when CopyAction returns ch will equal closing bracket}
```

```
577. var
578.    i: integer;
579.    done: boolean;
580.    posit: integer;  {integer value of n in $n}
581. begin with production[CardProd] do begin
582.    nextact := nextact+1;
583.    RhsArray[ThisRhs].CaseSelect := nextact;
584.    writeln(ActFile,'        ', nextact, ':  begin');
585.    while ch <> '}' do {copy routine}
586.       if ch = '$' then begin
587.          if eolninput then writeln(ActFile);
588.          readchar;
589.          if ch in ['0'..'9'] then begin {$n form}
590.             posit := 0;          {determine value of integer}
591.             while ch in ['0'..'9'] do begin
592.                posit := 10*posit + ord(ch)-ord('0');
593.                if eolninput then writeln(ActFile);
594.                readchar;
595.                end; {while}
596.             if posit > 0 then {number actually follows $}
597.                   {replace string in action routine}
598.                if posit > TotalVocab+1 then begin {illegal $n}
599.                   writeln(linecount:3, ' -- ',
600.                      '$', posit:2, ' refers to symbol not to ',
601.                      'its immediate right.');
602.                   ErrorFree := false
603.                   end
604.                else if posit = TotalVocab+1 then {refer to next vocab
                      symbol}
605.                   write(ActFile, 'LLStack[LLSentPtr+1].attribute')
606.                else begin {walk back to find right vocab symbol}
607.                   done := false;
608.                   i := 0;  {how far back we have walked}
609.                   while not done do begin
610.                       {walk back one node over for each symbol
611.                        following the one we want.}
612.                      i := i+1;
613.                      if RhsArray[ThisRhs-i].kind in
614.                            [nonterminal, literal, group] then
615.                      if RhsArray[ThisRhs-i].WhichVocabSymbol = posit then
616.                            done := true;
617.                      end; {while}
618.                   write(ActFile, 'LLStack[LLSentPtr-',i:2,'].attribute');
619.                   end {else begin}
620.             else {assign to attribute of lhs}
621.           writeln(ActFile,'LLStack[LLStack[LLSentPtr].parent].attribute')
622.             end {if ch in ['0'..'9']}
623.          else begin {just write $ and next char}
624.             write(ActFile, '$');
625.             write(ActFile, ch);
626.             writeln(linecount:3, ' -- ',
627.                'warning -- $ embedded in { ... } routine');
628.             if eolninput then writeln(ActFile);
629.             if ch = '{' then
630.                writeln(linecount:3, ' -- ',
631.                   'warning -- { embedded in { ... } routine');
632.             readchar;
633.             end; {else}
634.          end {if ch}
635.       else begin     {not a special character -- just copy}
636.          write(ActFile, ch);
637.          if eolninput then writeln(ActFile);
638.          if ch = '{' then
639.             writeln(linecount:3, ' -- ',
640.                'warning -- { embedded in { ... } routine');
641.          readchar;
642.          end; {else}
643.    if eolninput then writeln(ActFile);
644.    writeln(ActFile);
645.    writeln(ActFile, '        end; {', nextact, ' }');
646.    end; {with production}
647. end; {CopyAction}
648.
649.
650. procedure DoSynchronization;   {process synchronization information}
651. label 10;
652. var
653.    name: strings;       {token where synch takes place}
654.    posit: nonneg;       {position in production to recover to}
655. begin with production[CardProd] do begin
656.    readchar;          {skip past s}
657.    if TotalVocab >= 1 then
658.          {synch info does not increase CardRhs so CardRhs
659.           still has the value it had before synch info
660.           was encountered}
661.       if RhsArray[ThisRhs].kind in [nonterminal, group, literal] then
662.          RhsArray[ThisRhs].SynchIndex := CardSynch+1
663.       else begin
664.          ErrorFree := false;
665.          writeln(linecount:3, ' -- Synchronization info ',
666.             'must follow a vocabulary symbol.')
667.          end
668.    else {synch precedes all vocab symbols}
669.       ProdSynchIndex := CardSynch+1;
670.    repeat
671.       skipspace;
672.       fillid(name, spacers);
673.       CardSynch := CardSynch+1; {SynchData will be stored here}
674.       SynchData[CardSynch].TableIndex := OrderedFind(name);
675.       if SynchData[CardSynch].TableIndex = 0 then begin {undeclared}
676.          writeln(printable(name), ' is undeclared.');
677.          ErrorFree := false;
678.          end;
679.       skipspace;
680.       if ch <> '=' then begin
681.          name := AllBlanks;
682.          name[1] := ch;
683.          ErrorFree := false;
684.          writeln(printable(name), ' found but "=" expected
          after token name.');
685.          while ch <> '}' do readchar;
686.          goto 10;
687.          end
688.       else readchar; {skip past =}
689.       if ch <> '>' then begin
690.          name := AllBlanks; name[1] := ch;
691.          ErrorFree := false;
692.          writeln(printable(name), ' found but ">" expected after "="');
693.          while ch <> '}' do readchar;
694.          goto 10;
695.          end
696.       else readchar;
697.       skipspace;
698.       posit := 0;   {determine value of number}
699.       while ch in ['0'..'9'] do begin
700.          posit := 10*posit + ord(ch) - ord('0');
701.          readchar;
702.          end;
703.       if posit > 0 then begin
704.           {posit includes only nonterms and terms --
705.            TotalVocab has the current number seen}
706.          if (posit < TotalVocab) then
707.             begin {illegal to back up}
708.             ErrorFree := false;
709.             writeln(linecount:3, ' -- synchronization ',
710.                   'may not resume ',
711.                   'to the left of symbol where error occurs');
712.             end; {if posit <}
713.          SynchData[CardSynch].sent := posit
714.          end {if posit >}
715.       else if ch = '*' then begin  {skip past rest of rhs}
716.          SynchData[CardSynch].sent := maxint;
717.          readchar;
718.          end
719.       else begin
720.          ErrorFree := false;
721.          writeln(printable(name), ' found but "*" or integer expected');
722.          while ch <> '}' do readchar;
723.          goto 10;
724.          end;
725.       skipspace;
726.       if ch = ',' then readchar  {comma separates clauses}
727.       else if ch = ';' then begin  {"clean up" code follows}
728.          readchar; {skip past semicolon}
729.          CopyAction;  {copy code into LLact}
730.          if TotalVocab = 0 then {synch code begins rhs}
731.             ProdCaseSelect := nextact;
732.          end
733.       else if ch <> '}' then begin
734.          ErrorFree := false;
735.          writeln(printable(name), 'found but comma expected
          after clause.');
736.          while ch <> '}' do readchar;
737.          goto 10;
738.          end
739.    until ch = '}';
740.    10: readchar;   {skip past closing brace}
741.    end; {with production}
742.    CardSynch := CardSynch+1;      {add closing synch data}
743.    SynchData[CardSynch].TableIndex := 0;
744.    SynchData[CardSynch].sent := 0;
745. end; {DoSynchronization}
746.
747.
748. procedure DoSpecialCode; {process an action, patch, or synch routine}
749. begin
750.    readchar;  {skip past open bracket}
751.    if ch = 's' then
752.       DoSynchronization
753.    else begin {patch or action}
754.       with production[CardProd] do
755.          if ThisRhs = RhsSize then begin
756.             writeln(linecount:3, ' -- more right-hand side',
757.                ' elements in productions than limit -- ', RhsSize:4);
758.       writeln('       Recompile "generate.p" after increasing RhsSize');
759.             goto 1000;
760.             end
761.          else begin
762.             CardRhs := CardRhs+1;
763.             ThisRhs := ThisRhs+1;
764.             with RhsArray[ThisRhs] do
765.                if ch = 'p' then begin     {patch for synctactic error}
766.                   kind := patch;
767.                   readchar;         {skip past p}
768.                   end {if ch}
769.                else if ch = 'a' then begin {normal action code}
770.                   kind := action;
771.                   readchar;
772.                   end {else if ch}
773.                else begin
774.                   writeln(linecount:3, ' -- illegal specifier "',
775.                      ch, '" in {..} code. Assume it is action code.');
776.                   kind := action;
777.                   end; {with}
```

```
778.             CopyAction;
779.             readchar;  {skip past closing brace}
780.             end; {else}
781.    end; {else}
782. end; {DoSpecialCode}
783.
784.
785. procedure DoSelectionSet; {additional selection set info is processed}
786.       {Zuse automatically computes selection set for
787.        each production.  However, user can add other
788.        elements for error processing such as "any", or
789.        he can tell Zuse how to resolve selection set
790.        conflicts.}
791. var
792.    name: strings;  {selection set element}
793.    where: nonneg;  {location of name in SymbolTable}
794. begin
795.       {cursor points to '%' which signals beginning of selection set info}
796.    readchar;  {skip past '%'}
797.    skipspace;
798.    while ch <> ';' do with production[CardProd] do begin
799.       fillid(name, spacers+[';']);
800.       where := OrderedFind(name);
801.       if where > 0 then {element has been declared}
802.          if SymbolTable[where].kind in [literal, group] then
803.                   {element is a terminal}
804.             resolve := resolve + [where]
805.                  {add element to resolve which ensures this element
806.                  will end up in selection set of this production and
807.                  not in selection set of alternative productions}
808.          else begin  {element is not legal selection set member}
809.             ErrorFree := false;
810.             writeln(printable(name), ' should be a terminal.')
811.             end
812.       else begin  {element has not been declared}
813.          ErrorFree := false;
814.          writeln(printable(name), ' is undeclared.');
815.          end;
816.       skipspace;
817.       end; {with}
818. end; {DoSelectionSet}
819.
820.
821. begin with production[CardProd] do begin {DoRightHandSide}
822.    resolve := [];  {no resolvants until found on rhs of production}
823.    CardSel := 0;    {selection set elements not computed yet}
824.    TotalVocab := 0;  {no nonterms or terms seen yet}
825.         {haven't seen synch code applicable to entire rhs yet.}
826.    ProdSynchIndex := 0;
827.    ProdCaseSelect  := 0;
828.    rhs := ThisRhs+1; {begin storing rhs elements after last
829.                      element of previous production}
830.    CardRhs := 0;    {no rhs elements yet}
831.    while ch <> ';' do begin  {haven't reached end of rhs yet}
832.       if ch = '{' then        {action, patch, or synch routine}
833.          DoSpecialCode
834.       else if ch = '%' then {selection set info}
835.          DoSelectionSet
836.       else if ThisRhs < RhsSize then begin   {normal symbol}
837.          CardRhs := CardRhs+1;     {another rhs element}
838.          ThisRhs := ThisRhs+1;
839.          TotalVocab := TotalVocab+1;     {another term or nonterm}
840.          with RhsArray[ThisRhs] do begin
841.             WhichVocabSymbol := TotalVocab;
842.             fillid(name, spacers+[';']);
843.                   {presume no synch info until found}
844.             SynchIndex := ProdSynchIndex;
845.                   {presume no synch clean up code either}
846.             CaseSelect: = ProdCaseSelect;
847.             TableIndex := OrderedFind(name);
848.             if TableIndex > 0 then
849.                kind := SymbolTable[TableIndex].kind
850.             else begin
851.                ErrorFree := false;
852.                writeln(printable(name), ' is undeclared.');
853.                kind := nonterminal; {treat as a nonterminal since this
854.                                     is most general vocabulary class.}
855.                end
856.             end; {with}
857.          end {else}
858.       else if ThisRhs = RhsSize then begin   {prods too long}
859.          writeln(linecount:3, ' -- number of right-hand side',
860.             ' elements in productions longer than limit -- ', RhsSize:4);
861.          writeln('        Recompile "generate.p" after increasing RhsSize');
862.          goto 1000;
863.          end;
864.       skipspace;     {skip to next rhs element}
865.       end; {while}
866.    readchar;      {skip past ';'}
867.    end; {with}
868. end; {DoRightHandSide}
869.
870.
871. begin {DoProductions}
872.    nextact := 0;   {no actions yet}
873.    CardSynch := 0; {no synch data yet}
874.    CardProd := 0;  {no productions yet}
875.    ThisRhs := 0;   {no rhs elements yet}
876.    writeln('All declarations processed -- ', CardSymbol:3, ' symbols.');
877.    skipspace;
878.    while (ch <> '%') or (NextChar <> '%') do begin  {another production}
879.       CardProd := CardProd+1;
880.       if CardProd > ProdSize then begin
881.          writeln(linecount:3, ' -- too many productions.');
882.          writeln(' Recompile "generate.p" with ProdSize increased.');
883.          goto 1000;
884.          end;
885.       production[CardProd].nullable := notsure;
886.       DoLeftHandSide;
887.       DoRightHandSide;
888.       skipspace;
889.       end; {while}
890. end;  {DoProductions}
891.
892.
893. begin {DoGrammar}
894.    LineLength := 0;   {initially line is empty}
895.    LineMarker := maxint;  {force the reading of the first input line}
896.    linecount := 0;              {no lines read yet}
897.    rewrite(ActFile, 'LLact.i');          (* UNIX *)
898.    writeln(ActFile, '    procedure LLTakeAction(CaseIndex: integer);');
899.                {set up for case stmt}
900.    writeln(ActFile, '       begin');
901.    writeln(ActFile, '          case CaseIndex of');
902.    writeln(ActFile, '           0:;');
903.    declarations;
904.    SortTable;  {sort the symbol table}
905.    DoProductions;
906.    for i := 1 to CardSymbol do with SymbolTable[i] do
907.       if (ProdStart = 0) and (kind = nonterminal) then begin
908.          writeln('Nonterminal ', value, ' does not appear on the');
909.          writeln('  left-hand-side of any production.');
910.          ErrorFree := false
911.          end;
912.    writeln(ActFile);
913.    writeln(ActFile, '   end; {case}');
914.    writeln(ActFile, '   end; {LLTakeAct}');
915. end; {DoGrammar}
916.
917.
918. procedure ComputeSelectionSets; {compute selection set of each production}
919.                     {store in select field of production}
920. type
921.    matrix = packed array[1..TableSize] of intset;
922. var
923.    timer: real;    {keep track of time used for computation}
924.    AllTerms: intset;  {set of all literals and groups -- the terminals}
925.    i,j: integer;  {loop counter}
926.    BDW: matrix;      {holds "begins directly with" relation originally --
927.                       eventually becomes "begins with" relation}
928.    IDEO: matrix;     {holds "is direct end of" relation originally --
929.                       eventually becomes "is end of" relation}
930.    IFDB: matrix;     {holds "is followed directly by" relation
931.                       which eventually becomes "is followed by"
932.                       and then "extended is followed by"}
933.
934.
935. procedure MatrixMult( var l, r, result: matrix);
936.       {multiply matrices l and r yielding result}
937. var
938.    i,j,k: integer;
939.    temp: intset;
940. begin
941.    for i := 1 to CardSymbol do {initially result is empty}
942.       result[i] := [];
943.    for j := 1 to CardSymbol do begin {j is the column index}
944.       temp := [];
945.       for k := 1 to CardSymbol do {build j-th column of matrix r}
946.          if j in r[k] then temp := temp + [k];
947.       for i := 1 to CardSymbol do {i is the row index}
948.          if (l[i]*temp) <> [] then
949.                {i-th row and j-th column yield non-empty product}
950.             result[i] := result[i] + [j];
951.       end; {for j}
952. end; {MatrixMult}
953.
954.
955. procedure FindNullable;  {compute nullable nonterminals and productions}
956. label 10;
957. var
958.    change: boolean;
959.    i,j: integer;     {loop counters}
960.
961. begin
962.    change := false;
963.    for i := 1 to CardProd do with production[i] do
964.       if CardRhs = 0 then begin
965.          SymbolTable[lhs].nullable := null;
966.          nullable := null;
967.          change := true
968.          end; {if CardRhs = 0}
969.    while change do begin   {add to list of nullables}
970.       change := false;   {must make a change each iteration}
971.       for i := 1 to CardProd do begin with production[i] do
972.          if nullable = notsure then  begin
973.             for j := rhs to CardRhs+rhs-1 do with RhsArray[j] do
974.                case kind of
975.                   group, literal: begin
976.                      nullable := never;
977.                      goto 10;
978.                      end; {group,literal}
979.                   action, patch: ;
980.                   nonterminal:
981.                      case SymbolTable[TableIndex].nullable of
```

```
982.                      never: begin
983.                         nullable := never;
984.                         goto 10;
985.                         end; {never}
986.                      notsure: goto 10;
987.                      null: ;
988.                      end; {case SymbolTable}
989.                 end; {case kind}
990.           nullable := null;
991.           SymbolTable[lhs].nullable := null;
992.           change := true;
993.           end; {if nullable}
994.        10: ;  {early exit for loop}
995.        end; {for i }
996.     end; {while change}
997. end; {FindNullable}
998.
999.
1000. function NotNullable(index: nonneg): boolean;
1001.       {is SymbolTable[index].value nullable?}
1002. begin with SymbolTable[index] do
1003.    case kind of
1004.       nonterminal: if nullable = null then NotNullable := false
1005.                    else NotNullable := true;
1006.       literal, group: NotNullable := true;
1007.       end; {case kind}
1008. end; {NotNullable}
1009.
1010.
1011. procedure ReflexTrans(var RT: matrix);
1012.       {compute reflexive transitive closure of RT}
1013. var
1014.    i,j,k: integer;
1015.    done: boolean;
1016.    temp: intset;
1017. begin
1018.    for i := 1 to CardSymbol do {make relation reflexive}
1019.       RT[i] := RT[i] + [i];
1020.    repeat {compute closure}
1021.       done := true; {assume no change until proven otherwise}
1022.       for j := 1 to CardSymbol do  begin {j is the column index}
1023.          temp := [];
1024.          for k := 1 to CardSymbol do {build j-th column of RT}
1025.             if j in RT[k] then temp := temp + [k];
1026.          for i := 1 to CardSymbol do begin {i is the row index}
1027.             if not (j in RT[i]) then
1028.                if (RT[i]*temp) <> [] then begin
1029.                   {i-th row and j-th column yield non-empty product}
1030.                   RT[i] := RT[i] + [j];
1031.                   done := false;
1032.                   end;
1033.             end; {for i}
1034.          end; {for j}
1035.    until done;
1036. end; {ReflexTrans}
1037.
1038.
1039. procedure ExtendedIsFollowedBy( var EIFB, IEO: matrix);
1040.       {add EndOfSource info to "is followed by" relation}
1041.       {presume EIFB is originally "is followed by"}
1042. var
1043.    i: integer;
1044.    where: integer;   {location of EndOfSource in SymbolTable}
1045.    temp: strings;    {EndOfSource padded with blanks}
1046. begin
1047.    temp := AllBlanks;
1048.    temp[1] := EndOfSource;
1049.    where := OrderedFind(temp);
1050.    for i := 1 to CardSymbol do
1051.          {if symbol is at the end of the axiom then EndOfSource
1052.           is at the end of symbol}
1053.       if axiom in IEO[i] then
1054.          EIFB[i] := EIFB[i] + [where];
1055. end; {ExtendedIsFollowedBy}
1056.
1057.
1058. procedure IsFollowedBy( var IFB, IEO, BW: matrix);
1059.       {x is followed by y if xy is a substring of some sentential form}
1060. var
1061.    temp: matrix;
1062. begin {IsEndOf x IsFollowedBy x BeginsWith}
1063.    MatrixMult(IEO, IFB, temp);
1064.    MatrixMult(temp, BW, IFB);
1065. end; {IsFollowedBy}
1066.
1067.
1068. procedure IsEndOf( var IEO: matrix);
1069.       { B is end of C if there is a derivation C => wB
1070.        in any number of steps}
1071. begin
1072.       {presume IEO is originally "is direct end of" relation}
1073.    ReflexTrans(IEO);
1074. end; {IsEndOf}
1075.
1076.
1077. procedure IsDirectEndOf( var IDEO: matrix);
1078.       { B is the direct end of C if there is a production
1079.         C -> wBz where z is nullable}
1080. label 10;
1081. var
1082.    i,j: integer;
1083. begin
1084.    for i := 1 to CardSymbol do {assume no direct end of SymbolTable[i]}
1085.       IDEO[i] := [];
1086.    for i := 1 to CardProd do with production[i] do begin
1087.       for j := rhs+CardRhs-1 downto rhs do with RhsArray[j] do begin
1088.             {search from right to left on rhs of production}
1089.          if RhsArray[j].kind in [nonterminal, group, literal] then begin
1090.             IDEO[RhsArray[j].TableIndex] :=
1091.                 IDEO[RhsArray[j].TableIndex] + [lhs];
1092.                    {no point searching past first non-nullable symbol}
1093.             if NotNullable(RhsArray[j].TableIndex) then goto 10;
1094.             end; {if RhsArray}
1095.          end; {for j}
1096.       10:
1097.       end; {for i}
1098. end; {IsDirectEndOf}
1099.
1100.
1101. procedure IsFollowedDirectlyBy( var IFDB: matrix);
1102.       {B is followed directly by C if there is a production
1103.        D -> wBzCx where z is nullable}
1104. label 10;
1105. var
1106.    i,j,k: integer;
1107. begin
1108.    for i := 1 to CardSymbol do {presume symbol is not followed by anything}
1109.       IFDB[i] := [];
1110.
1111.    for i := 1 to CardProd do with production[i] do
1112.       for j := rhs to rhs+CardRhs-1 do with RhsArray[j] do begin
1113.          if kind in [nonterminal, group, literal] then
1114.             for k := j+1 to rhs+CardRhs-1 do begin
1115.                if RhsArray[k].kind in [nonterminal, literal, group] then
                       begin
1116.                   IFDB[TableIndex] := IFDB[TableIndex] +
1117.                      [RhsArray[k].TableIndex];
1118.                         {don't search past first non-nullable symbol}
1119.                   if NotNullable(RhsArray[k].TableIndex) then goto 10;
1120.                   end; {if RhsArray}
1121.                end; {for k}
1122.          10:
1123.          end; {for j}
1124. end; {IsFollowedDirectlyBy}
1125.
1126.
1127. procedure BeginsDirectlyWith( var BDW: matrix );
1128.       {C begins directly with B if there is a production
1129.        C --> wBz where w is nullable}
1130. label 10;
1131. var
1132.    i,j: integer;
1133. begin
1134.    for i := 1 to CardProd do with production[i] do begin
1135.       for j := rhs to rhs+CardRhs-1 do with RhsArray[j] do
1136.          if kind in [nonterminal, group, literal] then begin
1137.             BDW[lhs] := BDW[lhs] + [TableIndex];
1138.             if NotNullable(TableIndex) then goto 10;
1139.             end; {if kind}
1140.       10:
1141.       end; {for i}
1142. end; {BeginsDirectlyWith}
1143.
1144.
1145. procedure BeginsWith( var BW: matrix);
1146.       {C begins with B if there is a derivation C ==> Bw
1147.        in any number of steps}
1148. begin
1149.    ReflexTrans(BW);
1150. end; {BeginsWith}
1151.
1152.
1153. procedure FirstSymbol( var FS: matrix);
1154.       {terminal b is a FirstSymbol of B if there is a derivation
1155.        C ==> bw in any number of steps}
1156. begin
1157.    for i := 1 to CardSymbol do
1158.       if SymbolTable[i].kind = nonterminal then
1159.          FS[i] := FS[i]* AllTerms
1160.       else
1161.          FS[i] := [i];
1162. end; {FirstSymbol}
1163.
1164.
1165. procedure FirstProd( var FS: matrix);
1166.       {terminal b is in FirstProd of production C --> w
1167.        if C ==> w ==> bz for sentential form bz in any
1168.        number of steps}
1169.
1170.       {store computed answers in "select" field of production
1171.        since this is part of the selection set info}
1172. label 10;
1173. var
1174.    i,j: integer;
1175. begin
1176.    for i := 1 to CardProd do with production[i] do begin
1177.          {presume selection set is empty}
1178.       select := [];
1179.          {add FirstSymbols of rhs elements up thru
1180.           first non-nullable element}
1181.       for j := rhs to rhs+CardRhs-1 do with RhsArray[j] do begin
1182.          if kind in [nonterminal, literal, group] then begin
1183.             select := select + FS[TableIndex];
1184.             if NotNullable(TableIndex) then goto 10;
```

```
1185.             end; {if kind}
1186.          end; {for j}
1187.       10:
1188.       end; {for i}
1189. end; {FirstProd}
1190.
1191.
1192. procedure HandleResolvants;  {give priority to resolvants in
      selection sets}
1193. label 10;
1194. var
1195.    i,j: integer;
1196. begin
1197.    for i := 1 to CardProd do with production[i] do
1198.       if resolve <> [] then begin
1199.          for j := SymbolTable[lhs].ProdStart to CardProd do
1200.             if production[j].lhs <> lhs then
1201.                goto 10
1202.             else
1203.                production[j].select := production[j].select-resolve;
1204.          10:
1205.          select := select + resolve;
1206.          end; {if resolve}
1207. end; {HandleResolvants}
1208.
1209.
1210. procedure HandleConflicts;  {report conflicts among alternative
1211.       productions and resolve conflicts as indicated by supplemental
1212.       selection set info stored in resolve}
1213. label 10;
1214. var
1215.    any: boolean;
1216.    i,j,k: integer;
1217.    intersect: intset;
1218. begin
1219.    any := false;
1220.    for i := 1 to CardProd do with production[i] do begin
1221.       for j := i+1 to CardProd do
1222.          if production[j].lhs = lhs then  begin
1223.             intersect := select*production[j].select;
1224.             production[j].select := production[j].select - intersect;
1225.             if intersect <> [] then begin {conflict!}
1226.                if not any then begin
1227.                   if PrintSelect then begin
1228.                      writeln(LLselect, 'There are selection set conflicts',
1229.                            ' as indicated:');
1230.                      writeln(LLselect);
1231.                      end;
1232.                   writeln('There are selection set conflicts as
                      indicated:');
1233.                   writeln;
1234.                   any := true
1235.                   end;
1236.                for k := 1 to CardSymbol do
1237.                   if k in intersect then begin
1238.                      if PrintSelect then
1239.                         writeln(LLselect, SymbolTable[k].value, ' in ',
1240.                            'prods at lines ',
1241.                            production[i].line : 3,
1242.                            ' and ', production[j].line: 3, ' with',
1243.                            ' left-hand side ', SymbolTable[lhs].value);
1244.                      writeln(SymbolTable[k].value, ' in ',
1245.                         'prods at lines ',
1246.                         production[i].line : 3,
1247.                         ' and ', production[j].line: 3, ' with',
1248.                         ' left-hand side ', SymbolTable[lhs].value);
1249.                      end;
1250.                end;
1251.             end {if production}
1252.          else goto 10; {no more alternatives}
1253.          10:
1254.          end; {for i}
1255.       if PrintSelect then begin
1256.          writeln(LLselect); writeln(LLselect); end;
1257. end; {HandleConflicts}
1258.
1259. begin {ComputeSelectionSets}
1260. timer := clock; {keep track of how much time to compute selection sets}
1261.                (* UNIX *)
1262.    writeln('Starting to compute selection sets -- ', CardProd:3,
1263.          ' productions.');
1264.    FindNullable;   {determine which symbols and prods are nullable
1265.                     -- store answers in SymbolTable and productions
1266.                     respectively}
1267.
1268.       {compute set of all terminals}
1269.    AllTerms := [];   {initially empty}
1270.    for i := 1 to CardSymbol do
1271.       if SymbolTable[i].kind  in [literal, group] then
1272.          AllTerms := AllTerms + [i];
1273.
1274.    for i := 1 to CardSymbol do {initially, BDW is empty}
1275.       BDW[i] := [];
1276.
1277.    BeginsDirectlyWith( BDW );
1278.    BeginsWith( BDW );
1279.    FirstSymbol( BDW );
1280.    FirstProd( BDW );
1281.
1282.       {at this point selection set for each production
1283.        already includes first of rhs as computed in
1284.        FirstProd -- just add Follow info if needed }
1285.
1286.    IsDirectEndOf(IDEO);   {computes "is direct end of" relation}
1287.    IsEndOf(IDEO);  {transforms "is direct end of" into "is end of"}
1288.    IsFollowedDirectlyBy(IFDB);
1289.    IsFollowedBy(IFDB, IDEO, BDW); {transforms "is followed directly
1290.          by" into "is followed by"}
1291.    ExtendedIsFollowedBy( IFDB, IDEO);  {adds info for EndOfSource
       marker to
1292.          "is followed by"}
1293.
1294.        {can now add Follow info to select field}
1295.    for i := 1 to CardProd do with production[i] do
1296.       if nullable = null then {add terminal symbols}
1297.          select := select + IFDB[lhs];
1298.    HandleResolvants;
1299.    HandleConflicts;
1300.    for i := 1 to CardProd do with production[i] do
1301.       for j := 1 to CardSymbol do
1302.          if j in select then
1303.             CardSel := CardSel+1;
1304.    writeln('Seconds CPU time to compute selection sets -- ',
1305.          (clock-timer)/1000.0 :5:2 );  {time is in milliseconds}
1306.          (* UNIX *)
1307. end; {ComputeSelectionSets}
1308.
1309.
1310. procedure SaveGrammar;  {save entire grammar on disk in LLgram.
1311.                          Print grammar with selection sets on LLselect
1312.                          if requested.}
1313. var
1314.    NumValue: integer;  {number of selection set or vocabulary elements
1315.                         on current line of LLselect.}
1316.    i,j: nonneg;     {loop counter}
1317.    NumberOfTerminals: nonneg;
1318.    WhichTerminal: array[1..TableSize] of integer; {which numbered
1319.             terminal SymbolTable[i] is}
1320. begin
1321.    rewrite(LLgram);
1322.          {write Symbol Table}
1323.    NumberOfTerminals := 0;
1324.    for i := 1 to CardSymbol do with SymbolTable[i] do
1325.       if kind in [literal, group] then begin
1326.             {write symbol and kind}
1327.          NumberOfTerminals := NumberOfTerminals+1;
1328.          LLgram^.table := value;
1329.          put(LLgram);
1330.          if kind = group then
1331.             LLgram^.table[1] := 'g'
1332.          else
1333.             LLgram^.table[1] := 'l';
1334.          put(LLgram);
1335.          WhichTerminal[i] := NumberOfTerminals;
1336.          end; {if kind}
1337.    writeln(ConstFile, '    LLTableSize = ', NumberOfTerminals:4, ';');
1338.          {now the productions}
1339.    if PrintSelect then begin {print heading of selection set report.}
1340.       writeln(LLselect, 'Grammar productions with selection sets added:');
1341.       writeln(LLselect);
1342.       writeln(LLselect, 'Prod #  Line #       Production');
1343.       writeln(LLselect);
1344.       end;
1345.    LLgram^.grammar := SymbolTable[axiom].ProdStart; put(LLgram);
1346.    for j := 1 to CardProd do
1347.       with production[j] do begin
1348.          if PrintSelect then begin
1349.             writeln(LLselect);
1350.             write(LLselect, j:3, line: 9, '       ',
1351.                   PrintString(SymbolTable[lhs].value), ' = ');
1352.             end; {if PrintSelect}
1353.          LLgram^.grammar := SymbolTable[lhs].ProdStart; put(LLgram);
1354.          LLgram^.grammar := CardRhs; put(LLgram);
1355.          NumValue := 0; {no vocab symbols on current line of LLselect yet}
1356.          for i := rhs to rhs+CardRhs-1 do with RhsArray[i] do begin
1357.             if PrintSelect then begin
1358.                if NumValue = 6 then begin
1359.                   writeln(LLselect);
1360.                   write(LLselect,'                       ');
1361.                   NumValue := 0
1362.                   end; {if NumValue}
1363.                NumValue := NumValue + 1;
1364.                if kind in [nonterminal, literal, group] then
1365.                   write(LLselect, PrintString(SymbolTable[TableIndex]
                      .value))
1366.                else if kind = action then
1367.                   write(LLselect, ' {a..} ')
1368.                else
1369.                   NumValue := NumValue - 1; {didn't print anything}
1370.                end;
1371.             case kind of
1372.                nonterminal: begin
1373.                   LLgram^.grammar := ord('n'); put(LLgram);
1374.                   LLgram^.grammar := SymbolTable[TableIndex].ProdStart;
1375.                   put(LLgram);
1376.                   LLgram^.grammar := CaseSelect; put(LLgram);
1377.                   LLgram^.grammar := SynchIndex; put(LLgram);
1378.                   end;
1379.                group: begin
1380.                   LLgram^.grammar := ord('g'); put(LLgram);
1381.                   LLgram^.grammar := WhichTerminal[TableIndex]; put
                      (LLgram);
1382.                   LLgram^.grammar := CaseSelect; put(LLgram);
1383.                   LLgram^.grammar := SynchIndex; put(LLgram);
```

```
1384.                   end;
1385.                literal: begin
1386.                   LLgram^.grammar := ord('l'); put(LLgram);
1387.                   LLgram^.grammar := WhichTerminal[TableIndex]; put(LLgram);
1388.                   LLgram^.grammar := CaseSelect; put(LLgram);
1389.                   LLgram^.grammar := SynchIndex; put(LLgram);
1390.                   end;
1391.                patch: begin
1392.                   LLgram^.grammar := ord('p'); put(LLgram);
1393.                   LLgram^.grammar := CaseSelect; put(LLgram);
1394.                   end;
1395.                action: begin
1396.                   LLgram^.grammar := ord('a'); put(LLgram);
1397.                   LLgram^.grammar := CaseSelect; put(LLgram);
1398.                   end;
1399.                end; {case}
1400.                end; {if}
1401.
1402.                    {write out selection set info}
1403.          if PrintSelect then begin
1404.             writeln(LLselect);
1405.             write(LLselect, '             %');
1406.             end;
1407.          LLgram^.grammar := CardSel; put(LLgram);
1408.          NumValue := 0;  {how many selection set elements printed on
1409.                           current line of LLselect}
1410.          for i := 1 to CardSymbol do
1411.             if i in select then begin
1412.                LLgram^.grammar := WhichTerminal[i]; put(LLgram);
1413.                if PrintSelect then begin
1414.                   if NumValue = 6 then begin
1415.                      writeln(LLselect);
1416.                      write(LLselect, '                     ');
1417.                      NumValue := 0;
1418.                      end; {if NumValue}
1419.                   NumValue := NumValue + 1;
1420.                   write(LLselect, PrintString(SymbolTable[i].value));
1421.                   end; {if PrintSelect}
1422.                end; {if i}
1423.          if PrintSelect then
1424.             writeln(LLselect, ' ;');
1425.          end; {with production}
1426.
1427.                    {write how large RhsArray is}
1428.       writeln(ConstFile, '  LLRhsSize = ', ThisRhs:4, ';');
1429.
1430.                    {write out synchronization info}
1431.       for i := 1 to CardSynch do begin
1432.          if SynchData[i].TableIndex <> 0 then
1433.             LLgram^.grammar := WhichTerminal[SynchData[i].TableIndex]
1434.          else
1435.             LLgram^.grammar := 0; put(LLgram);
1436.             LLgram^.grammar := SynchData[i].sent; put(LLgram);
1437.          end; {for i}
1438. end; {SaveGrammar}
1439.
1440.
1441. begin {main program}
1442.    ErrorFree := true;  {initially no errors in grammar}
1443.    for i := 1 to StringSize do
1444.       AllBlanks[i] := ' ';
1445.    rewrite(ConstFile, 'LLconst.i');            (* UNIX *)
1446.    writeln;
1447.    DoGrammar;
1448.    if ErrorFree then begin
1449.       ComputeSelectionSets;
1450.       SaveGrammar;
1451.       end
1452.    else
1453.       writeln('Selection sets not computed because of fatal error.');
1454.    writeln(ConstFile, '     LLProdSize = ', CardProd, ';');
1455.    writeln(ConstFile, '     LLSynchSize = ', CardSynch, ';');
1456.    writeln(ConstFile, '     LLStringLength = ', StringSize, ';');
1457. 1000:
1458.    writeln;
1459. end.
```

Announcements

1 6032 PROCESSOR GETS PASCAL, PL/1, FORTRAN, AND C COMPILERS

Globally optimizing, commercial-grade compilers for PL/1, Fortran, Pascal, and C languages are now available for National Semiconductor's 16032 microprocessor. The units are available both as cross-compilers and as native compilers running under Bell Laboratories' Unix operating system—though they may easily be retargeted to other operating systems.

Translation Systems Inc. claims that tests of its Pascal programs by an independent telecommunications firm found a code density of 1.4, compared with 1.7 for National's and 2.2 for Motorola's. Theoretically, a proficient assembler programmer would write a code size of 1.

The company is currently offering original-equipment manufacturers distribution licenses for a one-time fee of $97,500, plus royalties. Products in the works include compilers for RPG-II, Cobol, Basic, and a full PL/1 implementation.

Tom Lindin, Translation Systems Inc.,
530 Atlantic Ave., Boston, MA 02210
Phone (617) 357-9433

'DBX'—NEW PRODUCT

Pascal & Associates of Chapel Hill, NC, announces a new product intended for PASCAL programmers who wish to develop data management applications. DBX stores keyed data-strings passed to it from a calling program. It is fast, simple, efficient, and inexpensive. It is written entirely in PASCAL. It is available — on disk — for the Apple II+/IIe and IBM PC microcomputers, Apple- and IBM-compatibles, and computers with standard 8″ SSSD disk format. Since DBX includes complete source listings, it can be entered and used on any computer with a PASCAL compiler.

DBX stores variable-length keyed strings of data in a disk file. It uses a modified ISAM (Indexed Sequential Access Method): It divides its keyed entries into "pages," and indexes the pages. In each page, the entries themselves are kept sorted. Each page consists of an exact number of disk blocks. This enables DBX to use high-speed fixed-length disk I/O routines. Thus, DBX can stores and retrieve entries very quickly, while limiting the average number of disk accesses to one per operation.

DBX also maintains memory dynamically; it minimizes — at run-time — the memory it uses. The number of files that DBX can simultaneously manipulate is limited only by the amount available memory.

DBX is modular; it comes as an "Intrinsic Unit" for inclusion in a UCSD p-System library. You can write a program to call it, or modify the calling program we've included as an example, or modify DBX itself. Calling programs pass a simple "callblock" containing input keys, input data, and operations; DBX returns output keys, output data, the output page, and a result code for the success or failure of the operation.

For $49.95, you get OBJECT AND SOURCE CODE for three programs:

- DBX itself;
- 'MINIBASE,' a simple example program that calls DBX to store, retrieve, or delete entries;
- 'DBTEST,' a diagnostic 'testbed' program that calls DBX, and displays all current input and output information.

You also get a 50-page manual that describes the data structures, procedures, and principal algorithms for all three programs, and gives many suggestions for modifying them for different purposes.

With no changes at all, MINIBASE is already an ideal user-oriented package for bibliographies, glossaries, inventory catalogs, and similar applications.

DBX is available only from Pascal & Associates, 135 East Rosemary Street, Chapel Hill, NC 27514. Send $49.95 plus $3.00 for shipping costs; or call 919-942-1411 — we take major credit cards.

PLUMB

LOUISVILLE, KY — Riverside Data Inc. introduces a new publication designed to help personal computer users get more work, fun and information out of their machines.

"PLUMB — Probing the World of Personal Telecommunications" is a newsletter to help computer users explore the many services available when the computer is connected with a modem and a telephone.

PLUMB contains information of interest to computer users, regardless of their level of expertise or the brand of computer they own.

Unlike a particular software program, telecommunications is a common denominator, allowing owners of all types of machines to communicate, share ideas and "homegrown" software, and obtain a wide range of sophisticated services.

The popularity of commercial timesharing services such as The Source and CompuServe is evidence of the growing interest in personal telecommunications.

Though most computer owners are aware of these services, there are hundreds of private and company-sponsored databases and systems that deliver a vast array of services free or at a minimal cost. PLUMB features these other services:

- Electronic mail
- Software sales and exchange
- Dating services and adult bulletin boards
- Financial and investment information
- Merchandise sales
- Online games

Some "underground" bulleting boards even deal with information to help crack protected software. PLUMB reports on new systems and features added to established systems. And it publishes reports designed to help computer users get what they want, faster and more efficiently.

Riverside Data's intention is to provide the home computer user with the kind of usable, non-technical information about telecommunications available nowhere else.

Computer magazines are filled with articles and advertisements about modems and communication programs, but very little information about what's available once the user is ready to go online.

One big attraction of telecommunications is its ability to serve as a common ground for all types of computers. Apple users can trade ideas with Atari users. Radio Shack owners can talk to IBM people. It's like an enormous festival for millions of computer owners.

PLUMB sells for $20 for the five issues published in 1983. Subscriptions should be mailed to: Riverside Data Inc., P.O. Box 300, Harrods Creek, KY 40027.

SCENIC COMPUTER INTRODUCES SPRINTER-2, A TEXT PROCESSOR FOR LARGE DOCUMENTS

SEATTLE, WA — SPRINTER-2, a new text processor has been developed by Scenic Computer Systems, Inc., specifically to meet the demands of producing books, reports, manuals and other large documents.

SPRINTER-2 frees the user from tedious and error-prone tasks such as compiling and verifying indices, tables of contents, lists of figures, and maintaining forward and backward references. Automatic numbering of chapters, sections, and pages reduces preparation time.

SPRINTER-2's built-in text formatting commands include automatic footnote placement and numbering, multicolumn formats, and powerful header and footer line capabilities.

According to Erik Smith, Scenic's President, "The real power of SPRINTER-2 lies in its Macro Formatting Language. The user can define macros (one word instructions) to carry out any sequence of the built-in commands and other macros."

The use of macros minimizes commands embedded in the text and assures consistency throughout a document. An entire document can be reliably reformatted in minutes simply by changing the macros definitions.

A text file can be printed without modification on any of the supported printers with any type style, since SPRINTER-2 does not rely on any printer specific features.

Since proofing a large document for spelling and typing errors is a major task, a spelling checker with an expandable 40,000 word dictionary is available.

SPRINTER-2 supports all popular daisy-wheel printers, including Diablo 630 and 1600 Series, NEC Spinwriters, Printmaster F-10, Starwriter F-10, Transtar 130 and 140, and Qume Sprint. Drivers for additional printers, phototypesetters and laser printers are under development.

SPRINTER-2 is written in Pascal and is available for any computer using SofTech Microsystems' p-System. This includes Corona, Compaq DEC PDP and LSI 11/23, IBM Personal Computer, NEC Advanced Personal Computer, Sge II and Sage IV, Scenic Model One, Texas Instrucments Professional Computer, and Victor 9000/Sirius 1.

SPRINTER-2 is priced at $350. The optional spelling checker is $125. Product literature or the 140-page users' manual can be obtained by contacting Scenic Computer Systems, Inc., 14852 NE 31st Circle, Redmond, WA 98052; (206) 885-5500.

NEWEST SAGE SUPERMICROS BROCHURE

Sage Computer Technology of Reno, NV, has just published a full-color brochure describing its new family of high-performance microcomputers.

The line has been expanded from one model to four, including the original Sage II in a brand new low-profile configuration.

A mere 3-7/8" from desk top to computer top, the low-profile Sage II packs the same power its predecessor is famous for — 2 million operations per second, on-board RAM expandability to 1/2 MByte, p-System, RAM-Disk, and one or two built-in low-profile floppy drives.

Three Sage IV models offer the same level of performance, plus RAM expandability to 1 MByte. Varying Winchester capacities and cabinet sizes distinguish the three.

The smaller Sage IV provides 10 MBytes of built-in Winchester capacity, while the largest boasts four built-in Winchesters totaling 200 MBytes. In each case, one or two 640 KByte floppy drives may be specified.

All four Sage computers typically load a 20K program from floppy diskette in about a second, according to a company spokesman. He said the unique Sage architecture, which permits the Motorola 68000 processor to handle data from the drives without skewing and interleaving, accounts for the remarkable loading speed.

The literature also includes a great deal of information about the standard operating system, the UCSD p-System. CP/M-68K, Modula 2 and Hyper-Forth operating systems are optionally available.

To receive a free copy of the brochure, write to Sage Computer Technology, 4905 Energy Way, Reno, NV 78502. The phone number is (702) 322-6868.

PROGRAMMING A PERSONAL COMPUTER IN EDISON

The book ***Programming a Personal Computer*** by Per Brinch Hansen has now been published. It describes the Edison System, a portable software system for small, personal computers, and illustrates how the principles of programming languages, compilers, operating systems and computer architecture are applied in the design of a complete software system.

The book includes the program text of an operating system and a compiler written in the programming language Edison and describes an instruction set tailored to the language. It also includes the text of a system kernel which interprets the portable Edison code.

Last September, a class of USC undergraduates used the book and the software to build working operating systems for IBM Personal Computers. They now hve the necessary practical background to appreciate the theoretical principles of operating systems.

The book is published by Prentice-Hall, Englewood Cliffs, NJ 07632. The Edison System is currently available for the Compaq Portable Computer and the LSI II Computer. The software is distributed by Professor Per Brinch Hansen, Computer Science Department, University of Southern California, Los Angeles, CA 90089.

CALL FOR PAPERS

SCS Conference on Simulation in Strongly Typed Languages ADA, PASCAL, SIMULA, . . .* This conference will provide a forum for presenting new approaches to developing, validating, and using simulation models. The focus will be on high level implementation languages. In particular, we solicit new approaches based on ADA, PASCAL, SIMULA, and other strongly typed languages. Accepted papers will appear in the conference Proceedings. Areas of interest include:

- Simulation in ADA, including discrete event, continuous, and combined approaches.
- Programming environments tailored to modeling and simulation, particularly those based on ADA, PASCAL, or SIMULA.
- Interactive model development and analysis, including interactive graphics tools and model specification languages.
- Novel hardwre and software architectures that support efficient model execution.
- The simulation of distributed computer systems and, in particular, software simulation and prototyping.

DEADLINES AND REQUIREMENTS

Papers should be no longer than 5,000 words, approximately 20 double-spaced pages in length, with author names appearing only on the cover paper. All papers will be refereed. Extended abstracts will also be considered. Submit five copies of the paper by 15 July 1983 to:

Dr. Ray Bryant
IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
(914) 945-3542

Authors of accepted papers will be notified by 1 September 1983. Camera-ready copies will be due no later than 1 November 1983.

CONFERENCE SITE

The conference will be held in beautiful San Diego, CA. This is your opportunity to combine an outstanding educational experience with a vacation from winter. Bring your family!

*This conference will be held as part of the SCS Multiconference, incorporating Modeling and Simulation on Microcomputers, Simulation in Strongly Typed Languages: ADA, PASCAL, SIMULA, . . ., and Simulation in Health Care Delivery Systems.

SCREEN EDITOR LETS USERS DEFINE FUNCTION KEYS, BOOSTS PRODUCTIVITY

User-definable function keys (macros) are available for the first time on a microcomputer screen editor that can run on all major hardware systems, including the IBM Personal Computer and the Apple II™, according to Volition Systems, developer of the software.

Called the Advanced System Editor (ASE), the software can be adapted to a variety of terminals and can increase productivity at the keyboard by at least 25 percent, according to Joel J. McCormack, company chairman.

ASE is now available for all versions of the UCSD Pascal™ system. It offers OEM's, software suppliers and end users important advances over the original screen-oriented editor, which as been a major strength of all UCSD Pascal development systems.

"ASE brings microcomputer users the text editing and program development resources usually associated with much larger computers," McCormack said. These include features that were not available on the original editor—the capability of editing very large files, function keys that can be easily trained, file selection by menu rather than human memory, the ability to edit a new file while still within another, and simplified keystroke sequences to accomplish the most common actions.

"Because all commands derive from arbitrary key sequences, it is easy to customize ASE to most keyboards, giving manufacturers and systems houses great flexibility in the choice of terminals," McCormack continued. In addition, he noted, ASE can take advantage of features such as the insert line capability that are found on more and more terminals today.

Application developers can define the keys they want their users to have because ASE comes with a separate configuration program enabling redefinition of commands or capabilities based on the user or application. "Coupled with macros tailored to a software developer's application packages, enhanced, more effective packages result," he said.

ASE is the only unbundled UCSD Pascal editor that is available without a full development system for distribution or incorporation into hardware and software systems.

The software is fully supported for all versions of the UCSD Pascal system. The IBM PC operates under UCSD Pascal systems as well as the Philips P2000, Digital's new Professional series, the Xerox 820, and all Texas Instruments minis and micros including the Business Systems 2000 and the Home Computer. Versions of UCSD Pascal run on all major microcomputer-based systems including those incorporating 8080's, Z-80's, LSI-II's, 6502's, 6800's, 6809's, 8086's, Z8000's, and 68000's. Apple Pascal is a UCSD Pascal derivative.

The Advanced System Editor was designed specifically to improve productivity at the keyboard during program development and text editing. It incorporates user-oriented conveniences not usually found in microcomputer editors.

"Because we are professional programmers with experience on a wide range of computers," McCormack said, "we included the features found on larger systems that we knew would speed work flow, and we eliminated bottlenecks and sources of frustration."

The overall result has been to increase the capacity of files that can be edited, automate repetitive tasks, reduce keystrokes, and boost the capabilities of the editor so that memorization is minimized and almost all work can be done without extra manipulation outside the editor.

File handling has been simplified immensely because the file size is limited by available disk space rather than by RAM memory size as was the case before, McCormack said. With ASE, a single file may fill an entire disk volume, so users are not forced to juggle split files.

User-defined function keys (macros), which are not available with the original UCSD Pascal screen editor, are another major benefit to users. These keys allow a user to automate tasks that recur within their particular program development or text editing environment.

Any sequence of keystrokes, including editor commands, can be "taught" to one of eight function keys. Once taught, pressing the function key, in effect, causes the same keystroke sequence to be repeated. Such macros are easy to use, but powerful enough for complex operations. Macros can, in fact, call other function keys, and they can also be used to make a change that affects an entire group of files.

The function key remembers the sequence taught to it until the editing session ends or the key is redefined by the user. Definitions can be saved in a terminal-independent fashion within the file being edited or within libraries of definitions.

"User-definable macros yield significant time savings for any task that must be done repetitively but selectively," McCormack noted.

In addition to user-definable function keys, Volition looked for other ways of reducing keystrokes and making keyboard time more productive.

"The screen-oriented editor had already done some keystroke optimization, but we took it much further," McCormack said.

Almost all moving commands are accomplished with just one keystroke. Furthermore, they can be used in exchange or delete modes, whereas before the moving commands were only available at the outermost edit level.

Single keystroke cursor positioning commands have been added to permit additional cursor movements such as moving word by word, moving backwards by a screen, moving to the beginning or end of a line, deleting by words, or returning to the home position. Variable tabs are also available.

ASE lets the user recall search or replacement strings with a single keystroke. The editor also enables the user to move portions of the text horizontally (opening or closing space). This feature enables users to move columns of data relative to each other.

Nested editing and menu selection of files are two important features that contribute to user satisfaction and reduce overhead. Now with ASE, both can be accomplished without having to leave the editor and go into the filer.

With the previous editor, memorization was required because files were listable only by the filer, and that showed only the file name, size and date last used. However, ASE is designed to reduce memorization. There is a menu available when entering the editor and when selecting files to be copied. The first line of each file is available as a memory jogger. Furthermore, a menu of the file markers is available.

"With ASE, menu selection makes the 'what file?' decision a one-character, multiple-choice answer by offering the selection of editable files and, if desired, the first lines of those files," McCormack said. Use of menus is not mandatory, however.

Nested editing lets users work on one file, leave it to work on another or to retrieve information from another, then "pop" back to the precise place where they were working in the original file. Nested editing sessions can descend to a depth of six files, disk space permitting. As a result, it is possible to edit numerous files and move text from one file to another without leaving the editor.

Other features of ASE not found in the original UCSD Pascal editor include extended, or "paint-mode," exchange, and change logging. With extended exchange, characters can be exchanged, inserted or deleted anywhere on the screen. It's particularly useful to users who want to create character graphics or to rule tables because instead of being restricted to left to right movement, they can now type in any direction they want.

Change logging allows the user to maintain a dated log of what was done in each editing session. That's an especially useful record to have if a number of programmers have worked on development of a program over a period of time. It is also useful for documents that have had multiple authors.

"Most users spend the majority of their time entering or altering text with an inadequate editor, and their productivity and satisfaction suffer," McCormack said. "We believe ASE is a vastly improved tool for text manipulation."

ASE is offered for distribution and sublicensing, with substantial discounts offered for quantity purchases. Telephone support is provided. It is immediately available from Volition Systems, P.O. Box 1236, Del Mar, CA 92014, (714) 457-3865, with single copies for evaluation priced at $175.00.

ASE object and source code are available.

ASE can be adapted for use on non-standard implementations of the UCSD Pascal system.

Volition Systems works to improve the productivity of computer users and the quality of their tools. It concentrates on systems software development and on software development and hardware design. Volition specializes in Pascal, Modula-2 and related software, and it has designed hardware architectures for high-level languages under contract to other companies.

TMUCSD Pascal is a trademark of the REgents of the University of California. Apple is a trademark of Apple Computer, Inc.

ASE™ — The Advanced System Editor™

Still using the standard screen editor?

If so, you're wasting your time! ASE, the Advanced System Editor, will significantly increase your editing throughput. ASE lets you:

- Reduce the number of keystrokes you have to type.
- Select files for editing without ever having to invoke the filer.
- Automate repetitive editing tasks with function keys, so you can sit back and relax while ASE does the editing for you.

If you aren't convinced that ASE is an absolute necessity, ask any ASE user or try it yourself. Once you've used ASE, you'll never want to go back!

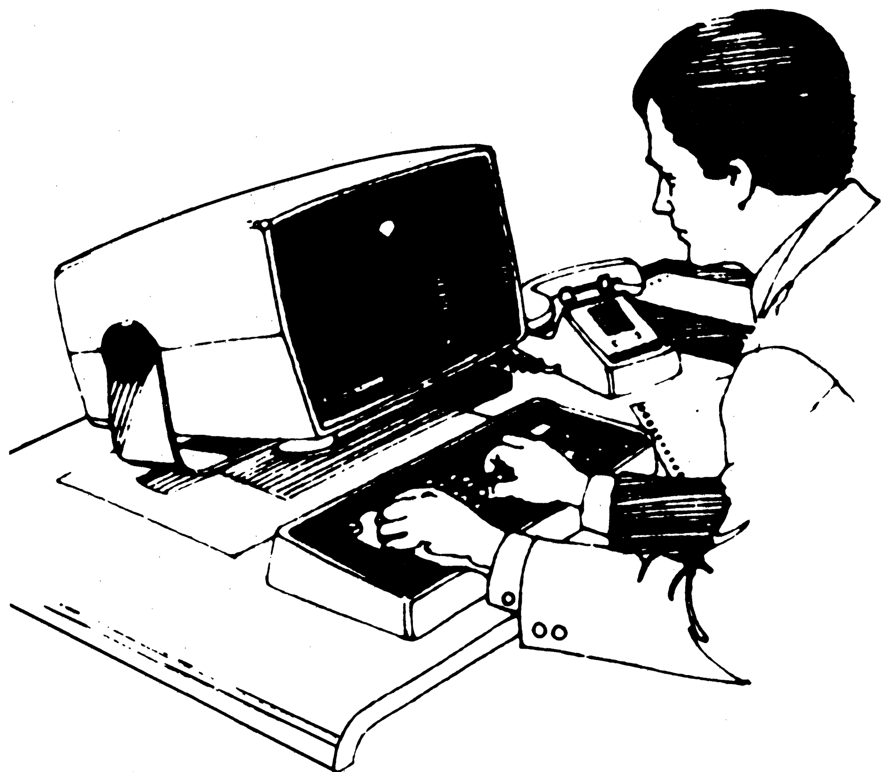
What is ASE?

ASE is a powerful text editor suitable for both word processing and programming environments. It incorporates many improvements into the standard UCSD Pascal screen editor. ASE features include:

1) **Large file editing:** ASE is not limited by memory size when editing text files; you can edit files as large as an entire disk volume. Source and destination files can be specified to reside on different disks.
2) **Keystroke optimization:** ASE introduces a number of single-keystroke commands which speed up editing: you can move to the next word, the next character occurrence, the beginning or end of a line, or the previous screen. Most moving commands can also be used in eX(change and D(elete, allowing you to delete to the next word, character occurrence, or end of the line.
3) **Extended exchange:** Exchange mode allows you to insert, delete, or exchange characters anywhere on the screen. Most edit commands can be invoked while in exchange mode. The typing direction in exchange mode can be changed to go up, down, or left, as well as right, making it easy to draw vertical lines and diagrams.
4) **Nested editing:** ASE allows you to edit another file in the middle of an edit session; when the nested edit session is finished, the editor "pops" back to the previous edit session. The copy buffer is saved across nested edit sessions, allowing you to easily move text from one file to another. Edit sessions can also be chained together, allowing you to edit a series of files without leaving the editor.
5) **File menu selection:** ASE displays a menu of available text files when you type "?" to a file name prompt. You no longer have to memorize all the files on your disks, or constantly flip between the filer and editor while editing a number of files. Typing "?" to the file menu prompt displays the first text line in each listed file, helping you to remember the contents of each file.
6) **Function keys:** Function keys can be 'trained' to remember a sequence of keystrokes; typing a function key causes it to automatically perform its routine. Function keys greatly simplify repetitive editing tasks; their power is limited only by your imagination.

ASE is available now for all versions of the UCSD Pascal system. A 100 page manual is included.

UCSD Pascal is a trademark of the Regents of the University of California. ASE and Advanced System Editor are trademarks of Volition Systems.



Modula 2

THE HISTORY OF THE DISER MODULA COMPUTER

1970 is an important date in the history of computers. It was then that Pascal was implemented and began its rise to enthusiastic acceptance by the programming world. This easy-to-learn, well defined language was created by Dr. Niklaus K. Wirth of Zurich, Switzerland.

Even as major universities, software houses, and computer companies were enhancing Pascal to suit their own particular needs, Wirth was assessing those needs and positing the next generation software development tool. The early stage of this new project was the creation of the language Modula. A research tool, Modula was not a general-purpose programming language, but rather a vehicle for Wirth to explore real-time control systems and for casting new light on the subject of assembly coding.

A year's sabbatical at Xerox Palo Alto Research Center (PARC), gave Wirth further insight into the concept of modules and high-level programming languages. He went back to Switzerland in 1977 to begin the project which has brought to the world the Modula Computer.

The research environment of the Institute fur Informatik at the ETH, Zurich, Switzerland provided the freedom to enter into the deisgn of not only a language, but concurrently the computer architecture best suited to implement the language. Initially, the language, a skillful blending of Pascal and Modula and called Modula-2, was compiled and run on a PDP-11. This was quickly followed, in 1979, by the first prototype of the new computer—then called the Lilith. By 1980, the Lilith had supported the software development Wirth had envisioned and ten Liliths were in use at the ETH.

Limited production of the Lilith began at Modula Research Institute in Provo, Utah. By 1981, twenty machines were in use with twenty more to follow in the year to come. These were situated primarily in the research and development areas of major universities and industries.

In addition to proving to be the ideal workstation for software development, 1981 saw the addition of a laser beam printer to the Lilith, the LPB-10. Extensive text editing and formatting was a natural for this system giving the user camera-ready, typeset quality copy from the printer.

A key component of this system was the Mouse. With three programmable buttons, many functions of text editing were greatly facilitated. Additional software, including bitmap graphics, coupled with the Mouse, supported various design functions.

It was time to bring this personal workstation to the marketplace. In January, 1983, the DISER Corporation was formed to begin full-fleged production of the computer, henceforth to be known as the MODULA COMPUTER.

MODULA-2

The language Modula-2—the notation in which this system presents itself to the software engineer—is designed as a total systems programming language. An assembler is not needed. The language is suited to both high-level programming in a machine-independent manner and low-level programming of machine-dependent aspects, such as device handling and storage allocation. The entire operating system, the compiler, the utility programs, and the library modules are programmed exclusively in Modula-2.

The compiler is subdivided into four parts. Each part processes the output of its predecessor in sequential fashion and is, therefore, called a pass. The first pass performs lexical and syntactic analysis, and it collects identifiers, allocating them in a table. The second pass processes declarations, generating the so-called symbol tables that are accessed in the third pass to perform the type-consistency checking in expressions and statements. The fourth pass generates code; its output is called M-code.

OPERATING ENVIRONMENT

The operating system is an "open" system. It is divided into three principle parts; the linking loader, the file system, and routines for keyboard input and text output to the display. The file system maps abstract files (sequences of words and characters) onto disk pages and provides the necessary basic routines for creating, naming, writing, reading, positioning, and deleting files. The loader and file system present themselves to the Modula-2 programmer as modules (packages) whose routines can be imported into any program. Whenever a program terminates, the basic operating system activates the command interpreter which requests the file name of the next program to be loaded and initiated.

The computer as "seen by the compiler" is implemented as a microprogrammed interpreter of the M-code. The M-code is designed with the principle goals of obtaining a high density of code and of making the process of its generation relatively systematic and straightforward. A high density of code is desirable not only n the interest of saving memory space, but also for reducing the frequency of instruction fetches. A comparison between two different, but strongly related compilers, revealed that M-code is shorter than code for the PDP-11 by a factor of almost four. This surprising figure is clear evidence of the inappropriate structure of conventional computer instruction sets, including those of most modern microprocessors that are still designed with the human assembly language coder in mind.

COMPUTER HARDWARE

The hardware consists of a central processing unit based on the Am2901 bit-slice processor, a multi-port memory with 128K words of 16 bits, a micro-code memory of 2K instructions implemented with PROMs, a controller each for the display, the disk, and interfaces for the keyboard, a cursor tracking device called the mouse, and an RS-232 serial line interface. The central processor operates at a basic clock cycle of 150 ns, the time required to interpret a micro-instruction. The most frequently occuring M-code instructions correspond to aobut 5 micro-instructions on the average.

The display is based on the raster scan technique using 832 lines of 640 dots each. Each of these 532,480 pixels (picture elements) is represented in main memory by one bit. If the entire screen is fully used, its bitmap occupies approximately 25% of memory. The display is refreshed through a 64-bit bus. The representation of each pixel in program accessible memory makes the display equally suitable for text, technical diagrams, and graphics in general.

In the case of text, each character is generated by copying the character's bitmap into the appropriate place of the screen's bitmap. This is done by software, supported by appropriate microcoded routines, corresponding to special M-code instructions. This solution, in contrast to hardwrae character generators, offers the possibility to vary the character's size, thickness (boldface), inclination (italics), and even style. In short, different fonts can be displayed.

This feature, which is particularly attractive for text processing, requires a substantial amount of computing power to be available in short bursts. The writing of a full screen, i.e. conversion of characters from ASCII code to correctly positioned bitmaps, takes about one fourth of a second. Using a small font, a full screen may display up to 11,000 characters.

The disk used is a Honeywell-Bull D-120 cartridge disk with a capacity of 10MBytes and a potential transfer rate of 720 kB/s resulting in an actual rate of 60kB/s for reading or writing of sequential files. Disk sectors, each containing 256 Bytes, are allocated in multiples of 8 on the same track. Allocation is entirely dynamic, and hence no storage contraction processes are needed to retrieve "holes."

The mouse is an input device that transmits signals to the computer which represent the mouse's movements on the table. These movements are translated (again by software) to a cursor displayed on the screen. The mouse also has three software-programmable buttons.

Reference: Wirth, N., *The Personal Computer Lilith, Report 40,* Institute fur Informatik, ETH Zurich, Switzerland; April 1981

Diser Corporation
P.O. Box 70
385 East 800 South
Orem, Utah 84057
Tel. 801-227-2300
Telex 453213

A COMPARISON OF MODULA-2 AND PASCAL

Modula-2—The Logical Successor to Pascal

In the years following the entrance of Pascal into the realm of computer languages, Pascal became increasingly appreciated for a number of reasons:

- clear definition of data structures and algorithms
- detection of syntax programming errors
- protection against illegal values being assigned to variables
- overcoming, in most instances, the need for subsets

Designed originally as a teaching language by Dr. Niklaus K. Wirth, the language tends to lead programmers to write well-structured programs. In addition, it is relatively easy to learn and essentially self-documenting.

As its popularity grew so did its "extensions". In particular, attempts to enhance real-time applications and I/O device handling. But these "solutions" created new problems.

While the rest of the world tried applying "fixes" to Pascal, Wirth analyzed the needs that had been identified and answered them with a complete language, Modula-2. The basic advancements fall into four categories:

1. module structure —
   Inherent management of complex programming problems occurs because Modula-2 supports structuring them as individual tasks.
2. separate compilation —
   The definition-module allows individual modules to be compiled separately without sacrificing error checking.
3. real-time primitives —
   Coroutines allow for real-time programming operations are defined in one or two separate modules. These modules are the only ones needing revision as a program is transported to another computer.
4. portability through encapsulated machine-dependent operations —
   Modula-2 programs are completely portable because machine-dependent programming operations are defined in one or two separate modules. These modules are the only ones needing revision as a program is transported to another computer.

In summary, Modula-2 is the logical successor to Pascal. it provides the solutions to Pascal's problems:

Pascal's Problems	Modula-2's Solutions
fixed arrays	open arrays
global variables only	local variables also
lack of separate compilation hindering large complex programs	separate compilation providing libraries of modules
rigid order of declarations	related declarations may be grouped
Boolean expressions are not conditionally evaluated	evaluations of conditions are ordered allowing branching upon satisfaction
limited I/O facilitation	standard library of I/O modules
does not allow low-level programming	controlled low-level access providing machine-level programming

VOLITION SYSTEMS

Dear fellow USUS Member:

You've probably had an experience like this:

• You spent two agonizing days hunting down a mysterious system bug. The cause? You changed a UNIT last week and forgot to recompile a program that uses it.

• You gave up trying to use your serial card with interrupts—it was too hard to program in assembly and too slow in Pascal.

• You like UCSD or Apple Pascal™, but have some doubts about it—everyone keeps saying the future belongs to UNIX™.

You're frustrated, yet you stick with UCSD. After all, look at the alternatives!

Anyway, that's how I felt before discovering **Modula-2, Niklaus Wirth's latest programming language.** *EE Times*, a leading electronics newsweekly, has called Modula-2 "the successor to Pascal." They're right.

With Modula-2, I'm more productive writing and maintaining code, and I can do things that just weren't possible with the Pascal system. Better yet, I'll be able to move my programs to other operating systems, thanks to the standard library Volition Systems provides with all its Modula-2 implementations.

Because you already know Pascal, **you'll be able to pick up Modula-2 in a matter of hours** and become proficient in less than a week. Furthermore, Modula-2 is based on the efficient II.0 architecture and is compatible with your existing UCSD Pascal and Apple Pascal software.

Sincerely,

Roger T. Sumner
Chief Programmer

UCSD Pascal is a trademark of the Regents of the University of California. UNIX is a trademark of Bell Laboratories. Apple is a trademark of Apple Computer, Inc. ASE is a trademark of Volition Systems.

What is Modula-2

The Modula-2 programming language was designed by Niklaus Wirth, Pascal's creator, as a simple but powerful alternative to assembly language, Pascal, 'C', and Ada. Modula-2 is easily learned by Pascal programmers, and it solves Pascal's problems in a consistent and structured fashion. Modula-2 language features include modules, concurrent processes, separate compilation, dynamic array parameters, and low-level machine access.

Modula-2 on UCSD Pascal

Modula-2 on UCSD Pascal is a software development system based on the version II UCSD Pascal system. The Modula-2 compiler accepts the full language with minor implementation restructions. Programs are compiled into P-code.

Separate compilation is fully supported, with up to 50 separately compiled modules per program. No linking is required—module binding is performed at run time. Modula-2 programs can call other programs as procedures. Interrupts are fully supported, allowing real-time programming in Modula-2. System-dependent library modules provide access to the UCSD Pascal file system and UCSD Pascal intrinsics. Standard library modules provide Modula-2 programs with a standard operating environment.

Standard Library

Standard library modules are implemented either as a stand-alone system or as an interface to an underlying operating system. Because all implementations present the same module interfaces, programs that use standard library modules are portable across all Modula-2 systems. Standard library facilities include:

1) **Console I/O:** The module InOut includes routines for reading and writing basic data types to the standard input and output files. Standard I/O defaults to the system console, but can be redirected to disk files. The module Terminal provides console input and output and keyboard polling.
2) **File I/O:** The modules Texts provides routines for reading and writing basic data types to text streams. The module Files includes routines for reading and writing byte streams and arbitrary data types to files. Random and sequential file access is supported. Directory operations allow programs to change disk file names or delete disk files.
3) **Storage management:** The module Storage includes routines for dynamic variable allocation and deallocation. Storage can allocate variable-sized buffers and indicate whether a given amount of storage is available.
4) **Program execution:** The module Program enables a Modula-2 program to call other programs as procedures. In addition to providing code overlays, this facility simplifies the construction of large software systems; major parts cn be written and tested as individual programs before being incorporated into the system as subprograms. Modula-2 programs and subprograms communicate by sharing library modules.
5) **Exception handling:** The modules Texts, Files, and Program include facilities for program control of run-time error handling and recovery.
6) **Process scheduling:** The module ProcessScheduler provides process scheduling and synchronization facilities via the type SIGNAL and the procedures WAIT and SEND.
7) **Strings:** The module Strings includes the string operations Insert, Delete, Pos, Concat, and Length.
8) **Decimal arithmetic:** The module Decimals provides decimal arithmetic and COBOL-style formatting routines suitable for business applications.
9) **Math functions:** The module MathLib0 includes the mathematical functions sin, cos, arctan, exp, ln, and sqrt.

System Components

The Modula-2 system includes a fast one-pass compiler, library manager utility, and a module library. Standard library modules provide I/O, program execution, storage allocation, strings, math functions, and decimal arithmetic. A copy of Niklaus Wirth's new book ***Programming in Modula-2*** is provided along with complete system documentation.

System-dependent Facilities

In addition to the standard module library, the Modula-2 system provides access to system-dependent facilities. On the Apple II and ///, special modules are provided to access Apple graphics and peripheral devices, and the interrupt system connects to the Apple hardware interrupt system. A utility program is also provided which can convert Apple Pascal intrinsic units into library modules.

On the Sage, a special library module is provided for performing 32-bit arithmetic, and the interrupt system connects to the event system defined in the Sage BIOS. Note that the Sage system includes UCSD Pascal-based system software and utilities.

Modula-2 versus UCSD Pascal™ — Seven Significant Differences

1) Modules versus units — Modula-2 divides its separately compiled modules into separate definition and implementation modules, allowing version control and easier library management. UCSD Pascal bundles a unit's interface and implementation parts into one compilation, preventing version control and making library management awkward. Modula-2's import/export statements and identifier qualification (e.g. "modulename.ident") provide explicit scope control over identifiers obrained from other modules. UCSD Pascal offers no identifier scope control between units. Finally, Modula-2 allows the declaration of local modules to improve the organization of compilation units. UCSD Pascal does not allow local units.

2) I/O and Storage Management — Modula-2 provides all I/O and storage management routines as library modules, allowing such routines to be redefined or removed from the runtime system. UCSD Pascal I/O and storage routines are hardwired into the operating system.

3) Concurrency — Modula-2 provides co-routines as the basic form of concurrency. Co-routines are simpler and faster than UCSD Pascal semaphores, and can also be used to construct most forms of process scheduling: rendezvous, message passing, signals, or semaphores.

4) Low-level Programming — Modula-2 provides explicit language features for low-level programming: type transfer functions, absolute-address variable declarations, bitsets, address arithmetic, and interrupt handling. In UCSD Pascal, low-level programming relies on assembly language or unsafe programming tricks which violate the Pascal language.

5) Parameters — Modula-2 provides procedure types: they are a generalization of Pascal's procedure parameter, and permit the definition of procedure parameters and procedure variables (a powerful concept new to most Pascalers). Modula-2 also provides open array parameters which accept arrays of any size. UCSD Pascal provides neither the procedure parameters nor conformant arrays defined in standard Pascal.

6) Statements & Expressions — Modula-2 provides a LOOP/EXIT statement, a FOR statement with arbitrary step values, and a CASE statement with an ELSE part and subranges allowed in case constants. Functions can return any type as a function result. Modula-2 defines short-circuit Boolean expression evaluation for simpler and more efficient programming. Constant expressions may appear anywhere a constant is allowed. Pascal provides none of these useful features.

7) Syntax — Modula-2 programs are more readable than Pascal. Identifiers are significant to any length (not just the first 8 characters). The INC and DEC procedures eliminate the need for 'i:=i+1' statements. There is no BEGIN/END statement, because structured statements (IF, WHILE) are terminated by END.

VOLITION INTRODUCES MODULA-2 FOR IBM PC

Niklaus Wirth's new programming language, Modula-2, is commercially available on the IBM Personal Computer for the first time, according to Volition Systems in Del Mar, CA.

Modula-2 comes as part of a complete software system based on a version II UCSD Pascal* operating system. The system, developed by Volition, features a fast, easy-to-use version of Modula-2 and works well even in the 64K IBM PC environment, a feat not achieved by other UCSD Pascal-based systems.

Wirth developed Modula-2 to overcome real-world deficiencies he recognized in Pascal, which he created earlier as a teaching language. The new language—designed to utilize standard software modules—offers great flexibility in the development of large, complex systems. It is paticularly suited for large industrial and commercial applications.

Volition's implementation of the new language offers a two-fold savings for software program developers using the IBM PC, according to Joel J. McCormack of Volition.

First, the use of standard software modules and separate compilation with automatic version control can save time and money during program development and maintenance.

"In addition, program developers will welcome Modula-2's portability," McCormack predicted, "because programs written in Modula-2 for the IBM PC are directly transferable to the Apple II system. In effect, they can double

their target market with a very minimal effort."

Volition's Modula-2 system for the PC includes a comprehensive module library, Modula-2 compiler, and tutorial programs designed to bring Pascal programmers up to speed on Modula-2 in a matter of hours.

The implementation makes special provisions for the needs of software developers, presenting a nicely integrated development environment. The compiler and editor communicate with each other to reduce development time, modules are dynamically linked so there is no separate linkage process required, and friendly user interface and consistent prompts are provided.

All the attractive features of Modula-2 are provided: low-level machine access, real-time control, concurrent processes, and type-secure separate compilation with automatic version control.

Real number and transcendental mathematical support is provided directly by the 8087 numerics processor. Performance using the 8087 is considerably faster than it would be without it.

"Interrupt handling is fully supported—programmers can now write real-time or multitasking applications in Modula-2 instead of resorting to error-prone assembly language," McCormack said.

Volition's unique development in the implementation is the standard library, a collection of modules that offers facilities normally provided by an operating system. The library provides console I/O, random access files, disk directory operations, format conversion, strings, decimal arithmetic, storae management, program execution and process scheduling. The standard library provides a portable interface to the underlying operating system.

"With Modula-2, you can develop portable software systems that run without change on a number of different operating systems," McCormack said. "This should be of obvious interest to software developers faced with writing applications which must run on all of today's popular microcomputers."

In addition to the IBM PC system, Volition currently provides Modula-2 for the Apple II (under Apple Pascal*) and for the Apple *///* (under SOS) and as part of a complete software system for computers based on the 8080/Z80 and 68000 processors.

Modula-2 for the IBM PC is immediately available from Volition Systems. The complete Modula-2 system includes Pascal and Modula-2 compilers, module library, the Advanced System Editor (ASE), p-NIX command shell (that provides a UNIX-like programming environment), and a complete set of utility programs.

The system is priced at $595. Educational, retailer, and distributor discounts are available.

Volition Systems concentrates on systems software development and on research and development in hardware and software. Since the company was founded in 1980, it has been a leader in the implementation and dissemination of the Modula-2 language and other high level languages and in the design and development of advanced computer architectures.

MRI ANNOUNCES MODULA-2 COMPILER FOR IBM PC

IBM PC owners can buy a full Modula-2 compiler for only $40 from the Modula Research Institute, a nonprofit organization in Provo, Utah. MRI has adapted the compiler for the IBM PC from the original Modula-2 compiler developed by Niklaus Wirth at the Institute for Informatics of the ETH in Zurich. The IBM PC compiler generates intermediate M-code similar in concept to the P-code of the original Zurich Pascal compiler.

Although the 4-pass Modula-2 compiler requires more compile time than a single-pass Pascal compiler, it provides M-code that is 30% more compact than the p-code and executes at least 20% faster. Use of M-code makes it possible to use programming tools transported from Wirth's Lilith, an optimized programmer's workstation that directly executes M-codes and is also available from MRI. MRI is now developing an M-code-to-native-code translator for the IBM PC to optimize execution time on the IBM PC at the expense of code compactness.

The $40 compiler for the IBM PC runs under DOS 2.0, requires 128k of RAM and 2 floppy disk drives, and is distributed with sample programs on two single-sided IBM floppy disks. Source code for the compiler is available from MRI on IBM PC floppy or 9-track tape for an additional $160. MRI also has versions of the compiler for the 68000 and the PDP-11.

MODULA-2 SOFTWARE FROM VOLITION TO LAUNCH SPRINGER-VERLAG LINE

Volition Systems Modula-2 will be the first software offering from Springer-Verlag, the international publisher of scientific, technical and medical books and journals, according to Volition Systems in Del Mar, CA.

Springer-Verlag New York Inc. will be handling Volition Systems Modula-2 software packages for worldwide distribution through the publisher's traditional retail channels. Volition's implementation of Niklaus Wirth's new Modula-2 programming language will be available for the Apple II and *//e*, the IBM Personal Computer and the Sage II and IV computers.

Until now, Volition has concentrated on sales of Modula-2 to OEM's, systems houses, software developers and manufactueres. "This agreement will significantly expand our marketing effort at the retail level," according to Joel J. McCormack of Volition Systems.

"We expect the broader availability of Modula-2 software will spark additional interest in this superior new programming language, particularly in the academic, scientific and technical fields where Springer's titles are highly respected," he continued.

Modula-2 software will be distributed as part of Springer's Computer Science Software Project and will be available beginning in October. Also available from Springer is Niklaus Wirth's book *Programming in Modula-2*, which includes a complete description of the language.

Wirth created Modula-2 (from MODUlar LAnguage) to overcome the real-world deficiencies of Pascal, a language which he previously created. The new language uses modules to facilitate development and maintenance of large, complex software systems, making it especially useful in large industrial, commercial and scientific applications.

The Volition Systems Modula-2 is available as a complete software development system and includes a comprehensive set of standard library modules and utilities as well as tutorial programs, system documentation and Wirth's book. Springer-Verlag will package and provide support for the systems it sells through its distribution channels.

Volition Systems concentrates on systems software development and on research and development in hardware and software. Since the company was founded in 1980, it has been a leader in the implementation and dissemination of the Modula-2 language and other high-level languages and in the design and development of advanced computer architectures.

Springer-Verlag is a leading international scientific, technical and medical publisher with 185 science journals and 900 new titles released annually. Located in New York, Berlin, Heidelberg and Tokyo, it maintains a worldwide network of interlocking editorial, production, marketing and distribution centers and publishes reference works, original research and advanced texts.

GET MORE FROM YOUR PASCAL SYSTEM
.... JOIN **USUS** TODAY

USUS is the USER'S GROUP for the most widely used, machine-independent software system.

If you use UCSD Pascal*, Apple Pascal** or the UCSD p-System, **USUS** will link you with a community of users that share your interests.

USUS was formed to give users an opportunity to promote and influence the development of UCSD Pascal and the UCSD p-System and to help them learn more about their systems. **USUS** is non-profit and vendor-independent.

Members get access to the latest UCSD p-System information and to extensive Pascal expertise. In **USUS**, you have formal and informal opportunities to communicate with and learn from other users via:

- NATIONAL MEETINGS
- **USUS** NEWS AND REPORT
- ELECTRONIC MAIL
- SOFTWARE LIBRARY
- SPECIAL INTEREST GROUPS

*UCSD Pascal and the UCSD p-System are trademarks of the Regents of the University of California.
**Apple Pascal is a trademark of Apple Computer, Inc.

USUS MEMBERSHIP APPLICATION
(Please complete both sides)

I am applying for $25 individual membership ______
$500 organization membership ______
$_____ air mail service surcharge ______

Rates are for 12 months and cover surface mailing of the newsletter. If you reside outside North America, air mail service is available for a surcharge. It is as follows: $5.00 annually for those in the Caribbean, Central America and Columbia and Venezuela; $10.00 annually for those in South America, Turkey and North Africa; and $15.00 for all others. Check or money order should be drawn on a U. S. bank or U.S. office.

Name/Title ____________________

Affiliation ____________________

Address ____________________

____

Phone (_____) _____-________ TWX/Telex __________

Option: Do not print my phone number in **USUS** rosters ______
Option: Print only my name and country in **USUS** rosters ______
Option: Do not release my name on mailing lists ______

USUS MEMBERSHIP BENEFITS

* * * * *

- NATIONAL MEETINGS twice a year let you learn from experts and try out the newest products. Meetings feature hardware and software demonstrations, tutorials, technical presentations and information, reduced-cost software library access, special interest group (SIG) meetings, and a chance to query "major" vendors.

- **USUS** NEWS AND REPORT brings you news and information about your operating system four times a year. It contains technical articles and updates, library catalog listings, SIG reports, a software vendor directory and organizational news.

- ELECTRONIC MAIL puts **USUS** subscribers in touch with a nationwide network of users. CompuServe MUSUS SIG is for data bases and bulletin board communications. GTE Telemail accommodates one-to-one messages.

- SOFTWARE EXCHANGE LIBRARY offers an extensive collection of tools, games, applications, and aides in UCSD Pascal source code at nominal prices.

- SPECIAL INTEREST GROUPS zero in on specific problems, represent member interests with manufacturers.

For more information, contact: Secretary, **USUS**, P. O. Box 1148, La Jolla, CA 92038, USA.

Computer System:
_____ Z-80 _____ 8080 _____ PDP/LSI-11 _____ 6502/Apple _____ 6800 _____ 6809
_____ 9900 _____ 8086/8088 _____ Z8000 _____ 68000 _____ MicroEngine _____ IBM PC
Other ____

I am interested in the following Committees/Special Interest Groups (SIGs):

_____ Advanced System Editor SIG
_____ Apple SIG
_____ Application Developer's SIG
_____ Communications SIG
_____ DEC SIG
_____ File Access SIG
_____ Graphics SIG
_____ IBM Display Writer SIG
_____ IBM PC SIG
_____ Meetings Committee
_____ Modula-2 SIG
_____ NEC Advanced PC SIG
_____ Publications Committee
_____ Sage SIG
_____ Software Exchange Library
_____ Technical Issues Committee
_____ Texas Instruments SIG
_____ UCSD Pascal Compatability SIG

Mail completed application with check or money order payable to **USUS** and drawn on a U.S. bank or U.S. office, to Secretary, **USUS**, P.O. Box 1148, La Jolla, CA 92038, USA.

Formerly *Pascal News*

2903 Huntington Road
Cleveland, Ohio 44120

Please enter my ☐ New or ☐ Renew

membership in Pascal Users Group. I understand I will receive *"Pascal & Modula2"* whenever it is published in this calendar year.

Pascal & Modula2 should be mailed

1 year ☐ in USA $25 ☐ outside USA $35 ☐ AirMail anywhere $60
3 year ☐ in USA $50 ☐ outside USA $80 ☐ AirMail anywhere $125

(Make checks payable to:
"Pascal Users Group," drawn on USA bank in US dollars)

Enclosed please find US $ __________ on check number __________

(Invoice will be sent on receipt of purchase orders. Payment must be received before magazine will be sent. Purchase orders will be billed $10 for additional work.)

(I have difficulty reading addresses.
Please forgive me and type or print clearly.)

My address is:

NAME ______________________________

ADDRESS ______________________________

____

____

PHONE ______________________________

COMPUTER ______________________________

DATE ______________________________

☐ This is an address correction. Here is my old address label:

____

____

____

Formerly *Pascal News*

2903 Huntington Road
Cleveland, Ohio 44120

Back issues are requested and sent in sets.

$15 ☐ set 0 Issues 1...8 (January 1974—May 1977)
Out of Print

$25 ☐ set 1 Issues 9...12 (September 1977—June 1978)

$25 ☐ set 2 Issues 13...16 (December 1978—October 1979)

$25 ☐ set 3 Issues 17...20 (March 1980—December 1980)

$25 ☐ set 4 Issues 21...23 (April 1981 [mailed January 1982]—
September 1981 [mailed March 1982])

Requests from outside USA please add $5 per set.

All memberships entered in 1983 will receive issue 24 and all other issues published in that year. Make check payable to: "Pascal Users Group," drawn on USA bank in US dollars.

Enclosed please find US $ __________ on check number __________

(I have difficulty reading addresses.
Please forgive me and type or print clearly.)

My address is:

NAME ______________________________

ADDRESS ______________________________

____

____

PHONE ______________________________

COMPUTER ______________________________

DATE ______________________________

Joining Pascal User Group?

- Membership is open to anyone: Particularly the Pascal user, teacher maintainer, implementor, distributor, or just plain fan.
- Please enclose the proper prepayment (check payable to "Pascal User's Group").
- When you join PUG any time within a year: January 1 to December 31, you will receive *all* issues of *Pascal & Modula2* for that year.
- We produce *Pascal & Modula2* as a means toward the end of promoting Pascal and communicating news of events surrounding Pascal to persons interested in Pascal. We are simply interested in the news ourselves and prefer to share it through *Pascal & Modula2*. We desire to minimize paperwork, because we have other work to do.

Renewing?

- Please renew early (before November) and please write us a line or two to tell us what you are doing with Pascal, and tell us what you think of PUG and *Pascal & Modula2*.

Ordering Back Issues or Extra Issues?

- Our unusual policy of automatically sending all issues of *Pascal News* to anyone who joins within a year means that we eliminate many requests for back issues ahead of time, and we don't have to reprint important information in every issue—especially about Pascal implementation!
- Issues 1...8 (January, 1974—May 1977) are *out of print*.
- Issues 9...12, 13...16, & 17...20, 21...23 are available from PUG(USA) all for $25.00 a set.
- Extra single copies of new issues (current academic year) are: $10 each—PUG(USA).

Sending Material For Publication?

- Your experiences with Pascal (teaching and otherwise), ideas, letters, opinions, notices, news, articles, conference announcements, reports, implementation information, applications, etc. are welcome. Please send material single-spaced and in camera-ready (use a dark ribbon and lines 15.5 cm. wide) form.
- All letters will be printed unless they contain a request to the contrary.

Facts about Pascal, THE PROGRAMMING LANGUAGE:

Pascal is a small, practical, and general-purpose (but *not all-purpose*) programming language possessing algorithmic and data structures to aid systematic programming. Pascal was intended to be easy to learn and read by humans, and efficient to translate by computers.

Pascal has met these goals and is being used successfully for:

- teaching programming concepts
- developing reliable "production" software
- implementing software efficiently on today's machines
- writing portable software

Pascal implementations exist for more than 105 different computer systems, and this number increases every month. The "Implementation Notes" section of *Pascal News* describes how to obtain them.

The standard reference ISO 7185 tutorial manual for Pascal is:

> *Pascal — User Manual and Report* (Second, study edition)
> by Kathleen Jensen and Niklaus Wirth.
> Springer-Verlag Publishers: New York, Heidelberg, Berlin
> 1978 (corrected printing), 167 pages, paperback, $7.90.

Introductory textbooks about Pascal are described in the "Here and There" section of *Pascal News.*

The programming language, Pascal, was named after the mathematician and religious fanatic Blaise Pascal (1623-1662). Pascal is not an acronym.

Remember, Pascal User's Group is each individual member's group. We currently have more than 3500 active members in more than 41 countries.

2903 Huntington Road
Cleveland, Ohio 44120

Return to:

2903 Huntington Road
Cleveland, Ohio 44120

Return Postage Guaranteed   Address Correction Requested

BULK RATE
U.S. POSTAGE
PAID
WILLOUGHBY, Ohio
Permit No. 58

If the number on the mailing label in brackets is **not** [84] or higher,
it is time to renew for 1984. Please detach and mail the self-addressed card below.

Renew my subscription for 1984
at $25 for the year.

☐ Check Enclosed
☐ Bill Me

Address Change Below