Pascal Users Group

NUMBER 22 & 23

Pascal News

COMMUNICATIONS ABOUT THE PROGRAMMING LANGUAGE PASCAL BY PASCALERS

SEPTEMBER, 1981

Return to:
Pascal Users Group
P.O. Box 4406
Allentown, Pa. 18104-4406

Return postage guaranteed
Address Correction requested

POLICY: PASCAL NEWS (15-Sep-80)

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

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

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

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

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

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

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

* Pascal News is divided into flexible sections:

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

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

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

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

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

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

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

- - - - - - ALL-PURPOSE COUPON - - - - - - (15-Dec-81)

Pascal Users Group
P.O. Box 4406
Allentown, Pa. 18104-4406 USA

Note

- We will not accept purchase orders.
- Make checks payable to: "Pascal Users Group", drawn on a U.S. bank in U.S. dollars.
- Note the discounts below, for multi-year subscription and renewal.
- The U. S. Postal Service does not forward Pascal News.

- -

		USA	UK	Europe	Aust.
[ ] Enter me as a new member for:	[ ] 1 year	$10.	#6.	DM20.	A$8.
	[ ] 2 years	$18.	#10.	DM45.	A$15.
[ ] Renew my subscription for:	[ ] 3 years	$25.	#15.	DM50.	A$20.

[ ] Send Back Issue(s) _____

[ ] My new address/phone is listed below

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

[ ] Comments: _____

_____

_____

ENCLOSED PLEASE FIND: _____.____

CHECK no. _____

NAME _____

ADDRESS _____

_____

_____

PHONE _____

COMPUTER _____

DATE _____

JOINING PASCAL USERS GROUP?

Membership is open to anyone: Particularly the Pascal user, teacher, maintainer, implementor, distributor, or just plain fan.
Please enclose the proper prepayment (check payable to "Pascal User's Group"); we will not bill you.
Please do not send us purchase orders; we cannot endure the paper work!
When you join PUG any time within a year: January 1 to December 31, you will receive all issues of Pascal News for that year.
We produce Pascal News as a means toward the end of promoting Pascal and communicating news of events surrounding Pascal to persons interested in Pascal. We are simply interested in the news ourselves and prefer to share it through Pascal News. We desire to minimize paperwork, because we have other work to do.

American Region (North and South America) Join through PUG(USA).

European Region (Europe, North Africa, Western Asia): Join through PUG(EUR) Pascal Users Group, c/o Grado Computer Systems & Software, Weissenburgerstrasse 25, D-8000, Munchen 80, Germany.

United Kingdom Region : join through PUG(UK) : Pascal Users Group, c/o Shetlandtel, Walls, Shetland, ZE2 9PF, United Kingdom.

Australasian Region (Australia, East Asia - incl.India & Japan): PUG(AUS). Pascal Users Group, c/o Arthur Sale, Department of Information Science, University of Tasmania, Box 252C GPO, Hobart, Tasmania 7001, Australia. International telephone: 61-02-202374

RENEWING?

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

ORDERING BACK ISSUES OR EXTRA ISSUES?

Our unusual policy of automatically sending all issues of Pascal News to anyone who joins within a year means that we eliminate many requests for backissues ahead of time, and we don't have to reprint important information in every issue--especially about Pascal implementations!

Issues 1 .. 8 (January, 1974 - May 1977) are out of print.

Issues 9 .. 12, 13 .. 16, & 17 .. 20 are available from PUG(USA) all for $15.00 a set, and from PUG(AUS) all for $A15.00 a set.

Extra single copies of new issues (current academic year) are: $5.00 each - PUG(USA); and $A5.00 each - PUG(AUS).

SENDING MATERIAL FOR PUBLICATION?

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

Pascal News #22 & 23 September 1981 Index

APPLICATION FOR LICENSE TO USE VALIDATION SUITE FOR PASCAL

Name and address of reqestor: _____

(Company name if requestor is a company): _____

_____

Phone Number : _____

Name and address to which information should _____

be addressed (write "as above" if the same) _____

_____

_____

Signature of requestor: _____

Date: _____

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

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

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

In return, the copyright holders grant full permission to use the programs and documentation contained in the Validation Suite for the purpose of compiler validation, acceptance tests, benchmarking, preparation of comparative reports and similar purposes, and to make available the listings of the results of compilation and execution of the programs to third parties in the course of the above activities. In such documents, reference shall be made to the original copyright notice and its source.

Distribution Charge: $50.00

Make checks payable to ANPA/RI in US dollars drawn on a US bank.

Remittance must accompany application.

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

Source Code Delivery Medium Specification;

( ) 800 bpi, 9-track, NRZI, odd parity, 600' magnetic tape

( ) 1600 bpi, 9-track, PE, odd parity, 600' magnetic tape

( ) ANSI-STANDARD

a) Select Character Code Set:

( ) ASCII ( ) EBCDIC

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

( ) 40 ( ) 20 ( ) 10

( ) Special DEC System Alternates:

( ) RSX-IAS PIP Format (requires ANSI MAGtape RSX SYSGEN)

( ) DOS-RSTS FLX Format

Office Use Only

Signed _____

Date _____

Richard J. Cichelli

On behalf of A.H.J. Sale and R.A.Freak

Editor's Contribution

GOOFED AGAIN!

Yes as all you loyal Pensylvanians have noticed in the last issue of PN we managed to mess up the zip code of Allentown PA, and of course the USPS has come down on us like a ton of bricks! Please note that the zip is 18014 not 18170. It has been corrected in the new APC.

THE NEW APC

Speaking of the new APC, we have simplified it some more, and added current prices for the UK and Europe, and have modified the reverse side of the coupon to reflect the new foreign editors and their current addresses.

THE LATEST EUROPEAN SOLUTION

Speaking of the European editors, we have two new ones! One for the UK, and one for the Continent. Nick Hughes will be handling all business for Britain, and Hellmut Weber will be in charge of the European Region. Please see the APC for their addresses.

ON CALLING

Please restrict yourself to written correspondence when dealing with PUG. This is strictly a scholarly function. None of the editors (including myself) gets paid. All have a real job that pays their bills, and they owe their office hours to their employer. All PUG work is donated on their own time. So please write to the appropriate regional editor. It leaves a documentary trail that can be followed and handled as fast as we can. Honest!

COMBINED ISSUE

This is of course a combined issue. We are doing this to catch up and to beat the postal system and their high rates. If this upsets anyone we are sorry. We are doing our best.

ON BEING THE EDITOR

Anyone who is interested in being the new editor of PN should write to me at the main address (APC).

STANDARDS

Good news from the standard front! 7185.1 was approved by the international committee. More next issue from Jim Miner the Standards Editor.

THIS ISSUE

The highlight of ths issue is the long awaited (from last issue at least!) of Andrew Tanenbaum's EM1 compiler. I think it is really great. Tell us what you think! In the Here and There section Gregg Marshall has summarized the past few issues (15 .. 19) implementation notes. Thanx. In addition to the EM1 compiler, the Applications section includes an improved version of the subroutine "options", as well as a tree printing routine, and a set of routines to compress and expand text using Huffman codes. Good work! And finally the articles section has some fine contributions. Many people have asked (on the phone ... see above) about how the various CP/m compilers stack up. Now we have an answer. Also there is an article of the experiences of a novice teaching Pascal. From a geography teacher no less! And finally a probing article by Jonathan Yavner concerning problems with Pascal and some proposals for their solution.

Hope you like it.

Rick

Here and There With Pascal

Summary of **Implementations**

ALL	#15:101	Pascal I (Derived from Pascal S)
BESM-6	#15:107	
Burroughs B5700	#15:107	
Burroughs B6700/B7700 (MCP)	#19:113	
CDC 6000	#19:115	
CDC 6000	#15:108	
Cyber 70 and 170	#15:108	
DEC PDP-11	#19:115	UCSD Pascal
DEC PDP-11	#15:111	
DEC PDP-11	#15:112	UCSD Pascal
DEC PDP-11	#15:124	
DEC PDP-11 (RSTS)	#15:100	Pascal S
DEC PDP-11 (RSX-11M/IAS)	#17:86	
DEC PDP-11 (RSX-11M/RT-11)	#15:101	Concurrent Pascal
DEC PDP-11 (Unix)	#15:111	
DEC PDP-11 (Unix)	#15:100	Pascal E
DEC PDP-11 (Unix)	#15:103	Modula
DEC PDP-15	#15:124	
DEC VAX	#17:89	
DEC VAX (Unix)	#19:115	
DG Eclipse	#17:106	
DG Eclipse (AOS	#15:110	RDOS,DOS)
DG Eclipse (AOS)	#15:109	
DG Eclipse (RDOS)	#15:108	
DG Nova (AOS	#15:110	RDOS,DOS)
Digico Micro 16E	#15:113	
Facom 230-45S	#15:112	
General Electric GEC4082	#15:113	
Golem B (GOBOS)	#17:104	
HP 1000	#19:116	
Honeywell 6000 (GCOS III)	#15:113	
Honeywell Level 6	#15:113	
IBM 3033	#19:120	
IBM 360/370	#15:114	
IBM 360/370	#15:115	
IBM 370	#17:104	
IBM 370	#19:117	
IBM 370	#15:124	
IBM 370	#17:102	
IBM 370/303x/43xx	#19:117	
IBM Series 1	#19:116	
IBM Series 1	#15:114	
ICL 1900	#15:116	
Intel 8080/8085	#15:119	
Intel 8080/8085	#15:118	
Intel 8080/8085	#15:119	
Intel 8080/8085	#17:102	
Intel 8080/8085	#15:117	
Intel 8080/8085 (CP/M)	#17:105	
Intel 8080/8085 (TRS-80)	#15:100	
Intel 8080/8085 (Northstar)	#15:100	
Intel 8086	#15:119	
Intel 8086	#15:103	
MOS Tech 6502 (Apple)	#15:107	
Modcomp II and IV	#15:120	
Motorola 6800	#15:120	
Motorola 6800	#19:120	
Motorola 6800	#19:121	
Motorola 6800	#17:102	
Motorola 6800 (Flex)	#15:123	
Motorola 68000	#19:121	
Motorola 6809	#15:103	
Motorola 6809 (MDOS09)	#17:102	
Nord 10 and 100 (Sintran III)	#15:121	
Perkin-Elmer 3220	#15:122	
Perkin-Elmer 7/16	#15:121	
RCA 1802	#17:103	
RCA 1802	#15:122	
Siemens 7.748	#15:124	
Sperry-Univac V77	#15:124	
Texas Instruments 990	#17:101	
Texas Intruments 9900	#15:124	
Zilog Z-80	#15:124	
Zilog Z-80	#19:123	
Zilog Z-80	#15:124	
Zilog Z-80	#17:88	
Zilog Z-80	#17:104	
Zilog Z-80 (CP/M)	#17:103	
Zilog Z-80 (TRS-80)	#15:124	
Zilog Z-80 (TRS-80)	#19:124	
Zilog Z80	#15:118	
Zilog Z80	#15:119	
Zilog Z8000	#15:119	

EM1 COMPILER


```
#include "../h/local.h"
#include "../h/em1.h"

{ (c) copyright 1980 by the Vrije Universiteit, Amsterdam, The Nether-
  lands.  Explicit permission is hereby granted to universities to use
  or duplicate this program for educational or research purposes.  All
  other  use or duplication  by universities,  and all use or duplica-
  tion by other organizations is expressly prohibited  unless  written
  permission  has  been obtained from the Vrije Universiteit. Requests
  for such permissions may be sent to

        Dr. Andrew S. Tanenbaum
        Wiskundig Seminarium
        Vrije Universiteit
        Postbox 7161
        1007 MC Amsterdam
        The Netherlands

  Organizations wishing to modify part of this software for subsequent
  sale  must  explicitly  apply  for  permission.  The exact arrange-
  ments will be worked out on a case by case basis, but at  a  minimum
  will require the organization to include the following notice in all
  software and documentation based on our work:

        This  product  is  based  on  the  Pascal  system
      developed  by  Andrew  S.  Tanenbaum, Johan W. Stevenson
      and Hans van Staveren of the Vrije  Universiteit,  Amster-
      dam, The Netherlands.
}

{if next line is included the compiler is written in standard pascal}
{#define       STANDARD        1}

{if next line is included, then code is produced for segmented memory}
{#define       SEGMENTS        1}

{Author:        Johan Stevenson                 Version:        31}
{$1- : no source line numbers}
{$r- : no subrange checking}
{$a- : no assertion checking}
#ifdef STANDARD
{$s+ : test conformancy to standard}
#endif

program pem(input,em1,errors);
{ This Pascal compiler produces EM1 code as described in
  - A.S.Tanenbaum, J.W.Stevenson & H. van Staveren,
        "Description of a experimental machine architecture for use of
        block structured languages" Informatika rapport 54,
  A description of Pascal is given in
  - K.Jensen & N.Wirth,PASCAL user manual and report,Springer-Verlag.
  Several options may be given in the normal pascal way. Moreover,
  a positive number may be used instead of + and -. The options are:
        e:      interpret assertions (+)
        o:      C-type strings allowed (-)
        d:      type long may be used (-)
        f:      size of reals in words (2)
        i:      controls the number of bits in integer sets (16)
        l:      insert code to keep track of source lines (+)
        o:      optimize (+)
        p:      size of pointers in words (1)
        r:      check subranges (+)
        s:      accept only standard pascal programs (-)
        t:      trace procedure entry and exit (-)
        u:      treat '_' as letter (-)
}
{*************************************************************}
#ifdef STANDARD
label 9999;
#endif

const

{powers of two}
  t7            = 128;
  t8m1          = 255;
  t8            = 256;
  t14           = 16384;
  t15m1         = 32767;

{EM-1 sizes}
  bytebits      = 8;
  wordbits      = 16;
  wbm1          = 15;           {wordbits-1}
  minint        = -t15m1;
  maxint        = t15m1;
  maxintstring  = '0000032767';
  maxlongstring = '2147483647';

  bytesize      = 1;
  wordsize      = 2;
  addrsize      = wordsize;
  pnumsize      = wordsize;
  shortsize     = wordsize;
  longsize      = 4;
#ifdef SFLOAT
  floatsize     = 4;
#endif
#ifndef SFLOAT
  floatsize     = 8;
#endif

{Pascal sizes, for ptrsize, realsize and fhsize see handleopts}
  { EM-1 requires that objects greater than a single byte start at a
    word  boundary,  so their address is even.  Normally, a full word
    is also allocated for objects of a single byte.  This extra  byte
    is really allocated to the object, not only skipped by alignment,
    i.e. if the value false is assigned to a  boolean  variable  then
    both bytes are cleared.  For single byte objects is packed arrays
    or packed records, however, only one byte is allocated,  even  if
    the next byte is unused.  Strings are packed arrays.  The size of
    pointers is 2 by default, but can be changed at  runtime  by  the
    p-option.  Floating point numbers in EM-1 currently have size 4,
    but this might change in the future to 8.  The default can be
    overwritten  by  the f-option.  The routines involved with align-
    ment are 'even', 'address' and 'arraysize'.
  }
  boolsize      = bytesize;
  charsize      = bytesize;
  intsize       = shortsize;
  buffsize      = 512;
  maxsetsize    = 4096;         {t15 div bytebits}

{maximal indices}
  idmax         = 8;
  fnmax         = 14;
  smax          = 72;
  rmax          = 72;
  imax          = 10;

{opt values}
  off           = 0;
  on            = 1;

{for push and pop: }
  global        = false;
  local         = true;

{set bounds}
  minsetlst     = 0;
  maxsetlst     = 15;           {default}

{constants describing the compact EM1 code}
  MAGICLOW      = 172;
  MAGICHIGH     = 0;
  memerror      = 0;
  mesoptoff     = 1;
  mesvirtual    = 2;
  mesmreg       = 3;
  mesline       = 4;
  mesfloats     = 5;

{ASCII characters}
  tab           = 9;
  newline       = 10;
  hortab        = 11;
  formfeed      = 12;
  carret        = 13;

{miscellaneous}
  maxsg         = 127;          {maximal segment number}
  maxcharord    = 127;          {maximal ordinal number of chars}
  maxargc       = 13;           {maximal index in argv}
  rwlim         = 38;           {number of reserved words}
  spaces        = '         ';
  emptyform     = '           ';

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



```
type
{scalar types}
  symbols       (comma,semicolon,colon1,colon2,notsy,lbrack,ident,
                 intcst,charcst,realcst,longcst,stringcst,nilcst,minsy,
                 plussy,lparent,arrow,arraysy,recordsy,setsy,filesy,
                 packedsy,progsy,labelsy,constsy,typesy,varsy,procsy,
                 funcsy,beginsy,gotosy,ifsy,whilesy,repeatsy,forsy,
                 withsy,casesy,becomes,starsy,divsy,modsy,slashsy,
                 andsy,orsy,eqsy,neqsy,gtsy,geqsy,ltsy,
                 leqsy,insy,endsy,elsesy,untilsy,ofsy,dosy,
                 downtosy,tosy,thensy,rbrack,rparent,period
                );              {the order is important}
  chartypes     (lower,upper,digit,layout,tabch,
                   quotech,dquotech,colonch,periodch,lessch,
                   greaterch,lparentch,lbracech,
                                        {different entries}
                 rparentch,lbrackch,rbrackch,commach,semich,arrowch,
                   plusch,minch,slash,star,equal,
                                        {also symbols}
                 others
                );
  standpf=      (pread,preadln,pwrite,pwriteln,pput,pget,
                 preset,prewrite,pnew,pdispose,ppack,punpack,
                 pmark,prelease,ppage,phalt,
                                        {all procedures}
                 feof,feoln,fabs,fsqr,ford,fchr,fpred,fsucc,fodd,
                 ftrunc,fround,fsin,fcos,fexp,fsqrt,fln,farctan
                                        {all functions}
                );              {the order is important}
  libnames=     (ELH ,EFL ,CLS ,WDW ,           {input and output}
                 OPN ,GETX,RDI ,RDC ,RDR ,RBL ,RLE ,
                                        {on inputfiles}
                 CRE ,PUTX,WRI ,WBI ,WRC ,WSC ,WRS ,WSS ,WRB ,
                 WBB ,WRR ,WBR ,WRL, WSL, WRF ,WRZ ,WSZ ,WLB ,PAG ,
                                {on outputfiles, order important}
                 ABR ,RND ,SIN ,COS ,EXPX,SQT ,LOG ,ATN ,
                                        {floating point}
                 ABI ,ABL ,BCP ,BTS ,HEWX,SAV ,RST ,INI ,MLT ,
                 ASS ,GTO ,PAC ,WBP, DIS, ABZ, MDI, MDL
                                        {miscellaneous}
                );
  structforms=  (scalar,subrange,pointer,power,files,arrays,carray,
                 records,variant,tag);          {order important}
  structflags=  (speak,withfile);
  identflags=   (refer,used,assigned,noreg,semeset);
  idclass=      (types,konst,vars,field,carrband,proc,func);
  kindofpfs=    (standard,formal,actual,extra,forward);
  where=        (blck,rec,wreo);
  strkind=      (cst,fixed,pfixed,loaded,ploaded,indexed);
  tmpstruct=    (eq,subeq,ir,ri,il,li,lr,rl,es,se,noteq);
                                        {order important}

\ {subrange types}
    sgrange=    0..maxsg;
    idrange=    1..idmax;
    fnrange=    1..fnmax;
```

```
    rwrange=      0..rwlim;
    bytes=        0..tbm1;

  {pointer types}
    sp=   ^structure;
    ip=   ^identifier;
    lp=   ^lbl;
    bp=   ^blockinfo;
    np=   ^nameinfo;

  {set types}
    sos=          set of symbol;
    setofids=     set of idclass;
    formset=      set of structform;
    sflagset=     set of structflag;
    iflagset=     set of identflag;

  {array types}
    alpha =packed array[idrange] of char;
    fntype=packed array[farange] of char;

  {record types}
    errec=record
      errno:integer;                {error number}
      mess:alpha;                   {identifier parameter if required}
      mesi:integer;                 {numeric parameter if required}
      chno:integer;                 {column number}
      linc:integer;                 {line number}
      linr:integer;                 {relative to start of (included) file}
      orig:integer;                 {idem, but before preprocessing}
      fnam:fntype;                  {source file name}
    end;

    position=record                 {the addr info of certain variable}
      ad:integer;                   {for locals it is the byte offset}
      lv:integer;                   {the level of the beast}
#ifdef SEGMENTS
      sg:segrange                   {only relevant for globals (lv=0) }
#endif
    end;

  {records of type attr are used to remember qualities of
    expression parts to delay the loading of them.
    Reasons to delay the loading of one word constants:
          - bound checking
          - set building.
    Reasons to delay the loading of direct accessible objects:
          - efficient handling of read/write
          - efficient handling of the with statement.
  }
    attr=record
      exp:sp;                           {type of expression}
      packbit:boolean;                  {true for packed elements}
      ak:attrkind;                      {access method}
      pos:position;                     {sg, lv and ad}
      {if ak=cst then the value is stored in ad}
    end;

    nameinfo=record                 {one for each separate name space}
      nlink:np;                     {one deeper}
      fname:ip;                     {first name: root of tree}
      case occur:where of
        blok:();
        rec: ();
        wrec:(wa:attr)              {name space opened by with statement}
    end;

    blockinfo=record        {all info of the current procedure}
      nextbp:bp;            {pointer to blockinfo of surrounding proc}
      lc:integer;           {data location counter (from begin of proc) }
      ilbno:integer;        {number of last local label}
      forwcount:integer;    {number of not yet specified forward procs}
      lchain:lp;            {first label: header of chain}
    end;

    structure=record
      size:integer;                             {size of structure in bytes}
      sflag:sflagset;                           {flag bits}
      case form:structform of
        scalar  :(scalno:integer;               {number of range descriptor}
                  fconst:ip                     {names of constants}
                 );
        subrange:(min,max:integer;              {lower and upper bound}
                  rangetype:sp;                 {type of bounds}
                  subrno:integer                {number of subr descriptor}
                 );
        pointer :(eltype:sp);                   {type of pointed object}
        power   :(elset:sp);                    {type of set elements}
        files   :(filtype:sp);                  {type of file elements}
        arrays,carray:
                 (eltype:sp;                    {type of array elements}
                  inxtype:sp;                   {type of array index}
                  arpos:position                {position of array descriptor}
                 );
        records :(fstfld:ip;                    {points to first field}
                  tagsp:sp                      {points to tag if present}
                 );
        variant :(varval:integer;               {tag value for this variant}
                  nxtvar:sp;                    {next equilevel variant}
                  subtsp:sp                     {points to tag for sub-case}
                 );
        tag     :(fstvar:sp;                    {first variant of case}
                  tfldsp:sp                     {type of tag}
                 )
    end;

    identifier=record
      idtype:sp;                            {type of identifier}
      name:alpha;                           {name of identifier}
      llink,rlink:ip;                       {see enterid,searchid}
      next:ip;                              {used to make several chains}
      iflag:iflagset;                       {several flag bits}
```



```
    case klass:idclass of
      types   :();
      konst   :(value:integer);         {for integers the value is
                                         computed and stored in this field.
                                         For strings and reals an assembler constant is
                                         defined labeled '.1', '.2', ...
                                         This '.' number is then stored in value.
                                         For reals value may be negated to indicate that
                                         the opposite of the assembler constant is needed. }
      vars    :(vpos:position);         {position of var}
      field   :(foffset:integer);       {offset to begin of record}
      carrbnd :();                      {idtype points to carray}
      proc,func:
        (case pfkind:kindofpf of
           standard:(key:standpf);      {identification}
           formal,actual,forward,extrn:
             (pfpos:position;           {lv gives declaration level.
                                         sg gives instruction segment of this proc and
                                         ad is relevant for formal pf's and for
                                         functions (no conflict!!).
                                         for functions: ad is the result address.
                                         for formal pf's: ad is the address of the
                                         descriptor }
              pfno:integer;             {unique pf number}
              parhead:ip;               {head of parameter list}
              handle:integer            {le when heading scanned}
             )
        )
  end;

  lblrecord
    nextlp:lp;              {chain of labels}
    seen:boolean;
    lbval:integer;          {label number given by the programmer}
    lbname:integer;         {label number given by the compiler}
    lbdlb:integer           {zero means only locally used,
                             otherwise dlbno of label information}
  end;

{------------------------------------------------------------------------}
var  {the most frequest used externals are declared first}
  sy:symbol;                {last symbol}
  a:attr;                   {type,access method,position,value of expr}
{returned by insym}
  ch:char;                  {last character}
  chsy:chartype;            {type of ch, used by insym}
  val:integer;              {if lsmt symbol is an constant }
  iz:integer;               {string length}
  eol:boolean;              {true if current ch replaces a newline}
  inrestring:boolean;       {true for strings in ' '}
  id:alpha;                 {if last symbol is an identifier}
{some counters}
  line:integer;             {line number on code file (1..n) }
  dlbno:integer;            {number of last global number}
  lmax:integer;             {keeps track of maximum of lc}
  level:integer;            {current static level}
  ptrsize:integer;
  realsize:integer;
  fhsize:integer;           {file header size}
  argc:integer;             {index in argv}
  lastpfno:integer;         {unique pf number counter}
  oopt:integer;             {C-type strings allowed if on}
  dopt:integer;             {longs allowed if on}
  iopt:integer;             {number of bits in sets with base integer}
  aopt:integer;             {standard option}
{pointers pointing to standard types}
  realptr,intptr,textptr,emptyset,boolptr:sp;
  charptr,nilptr,stringptr,longptr:sp;
{flags}
  giveline:boolean;         {give source line number at next statement}
  including:boolean;        {no LIN's for included code}
  eofexpected:boolean;      {quit without error if true (nextch) }
  main:boolean;             {complete programme or a module}
  intypedec:boolean;        {true if nested in typedefinition}
  fltused:boolean;          {true if floating point instructions are used}
  seconddot:boolean;        {indicates the second dot of '..'}
{pointers}
  fwptr:ip;                 {head of chain of forward reference pointers}
  progp:ip;                 {program identifier}
  currproc:ip;              {current proc/func ip (see casestatement)}
  top:np;                   {pointer to the most recent name space}
  lastap:np;                {pointer to nameinfo of last searched ident }
{records}
  b:blockinfo;              {all info to be stacked at pfdeclaration}
  e:errec;                  {all info required for error messages}
  fa:attr;                  {attr for current file name}
{arrays}
  source:fntype;            {name of pascal source file}
  strbuf:array[1..smax] of char;
  iop:array[boolean] of ip;
                            {false:standard input, true:standard output}
  rw:array[rwrange] of alpha;
                            {reserved words}
  frw:array[0..idmax] of integer;
                            {indices in rw}
  rsy:array[rwrange] of symbol;
                            {symbol for reserved words}
  os:array[char] of chartype;
                            {chartype of a character}
  ssy:array[rparentoh..equal] of symbol;
                            {symbol for single character symbols}
  lmn:array[libmnem] of packed array[1..4] of char;
                            {mnemonics of pascal library routines}
  opt:array['a'..'z'] of integer;
  forceopt:array['a'..'z'] of boolean;
                            {26 different options}
  undefip:array[idclass] of ip;
                            {used in searchid}
  argv:array[0..maxargs] of
        record name:alpha; ad:integer end;
                            {save here the external heading names}
{files}
```

```
449     em1:file of byte;     {the EM1 code}
450     errors:file of errec;
451                           {the compilation errors}
452   {*************************************************************}

454   procedure gen2bytes(b:byte; i:integer);
455   var b1,b2:byte;
456   begin
457     if i<0 then
458       if i<minint then begin b1:=0; b2:=t7 end
459       else begin i:=-i-1; b1:=t8m1 - i mod t8; b2:=t8m1 - i div t8 end
460     else begin b1:=i mod t8; b2:=i div t8 end;
461     write(em1,b,b1,b2)
462   end;

464   procedure genost(i:integer);
465   begin
466     if (i>=0) and (i<sp_nost0) then write(em1,i+sp_fost0)
467     else gen2bytes(sp_ost2,i)
468   end;

470   procedure genlb(i:integer);
471   begin if i<t8 then write(em1,sp_ilb1,i) else gen2bytes(sp_ilb2,i) end;

473   procedure genilb(i:integer);
474   begin lino:=lino-1;
475     if i<sp_nilb0 then write(em1,i+sp_filb0) else genlb(i);
476   end;

478   procedure gendlb(i:integer);
479   begin if i<t8 then write(em1,sp_dlb1,i) else gen2bytes(sp_dlb2,i) end;

481   procedure gen0(b:byte);
482   begin write(em1,b); lino:=lino+1 end;

484   procedure gen1(b:byte; i:integer);
485   begin gen0(b); genost(i) end;

487   procedure gend(b:byte; d:integer);
488   begin gen0(b); gendlb(d) end;

490   procedure genident(nametype:byte; var s:alpha);
491   var i,j:integer;
492   begin i:=idnam;
493     while (s[i]=' ') and (i>1) do i:=i-1;
494     write(em1,nametype,i);
495     for j:=1 to i do write(em1,ord(s[j]))
496   end;

498   procedure gensp(m:libsmnem);
499   var i:integer;
500   begin gen0(op_eal); write(em1,sp_pnam,4);
501     for i:=1 to 4 do write(em1,ord(lms[m][i]))
502   end;

504   procedure genpnam(b:byte; fip:ip);
505   var n:alpha; i,j:integer;
506   begin
507     if fip^.pfpos.lv<=1 then n:=fip^.name else
508       begin n:='          '; j:=1; i:=fip^.pfno;
509         while i<>0 do
510           begin j:=j+1; n[j]:=chr(i mod 10 + ord('0')); i:=i div 10 end;
511       end;
512     gen0(b); genident(sp_pnam,n)
513   end;

515   procedure genend;
516   begin write(em1,sp_cend) end;

518   procedure genlin;
519   begin giveline:=false;
520     if opt['l']<>off then if main then gen1(op_lin,e.orig)
521   end;

523   procedure genreg(ad,sz,nr:integer);
524   begin
525     if sz<=wordsize then
526       begin gen1(ps_mes,mesreg); genost(ad); genost(nr); genend end
527   end;

529   {*************************************************************}

531   procedure puterr(err:integer);
532   {as you will notice, all error numbers are preceded by '+' and '0' to
533     ease their renumbering in case of new errornumbers.
534   }
535   begin e.erno:=err; write(errors,e);
536     if err>0 then begin gen1(ps_mes,meserror); genend end
537   end;

539   procedure error(err:integer);
540   begin e.mess:=spaces; e.mesi:= -1; puterr(err) end;

542   procedure errid(err:integer; var id:alpha);
543   begin e.mess:=id; e.mesi:= -1; puterr(err) end;

545   procedure errint(err:integer; i:integer);
546   begin e.mesi:=i; e.mess:=spaces; puterr(err) end;

548   procedure asperr(err:integer);
549   begin if e.asp<>nil then begin error(err); e.asp:=nil end end;

551   procedure teststandard;
552   begin if sopt<>off then error(-(+01)) end;

554   procedure enterid(fip: ip);
555   {enter id pointed at by fip into the name-table,
556     which on each declaration level is organised as
557     an unbalanced binary tree}
558   var nam:alpha; lip,lip1:ip; lleft,again:boolean;
559   begin nam:=fip^.name; again:=false;
560     lip:=top^.fname;
```



```
561     if lip=nil then top^.fname:=fip else
562       begin
563         repeat lip1:=lip;
564           if lip^.name>nam then
565             begin lip:=lip^.llink; lleft:=true end
566           else
567             begin if lip^.name=nam then again:=true;  {name conflict}
568               lip:=lip^.rlink; lleft:=false;
569             end;
570         until lip=nil;
571         if lleft then lip1^.llink:=fip else lip1^.rlink:=fip
572       end;
573     fip^.llink:=nil; fip^.rlink:=nil;
574     if again then errid(+02,nam);
575   end;

577   procedure initpos(var p:position);
578   begin p.lv:=level; p.ad:=0;
579   {ifdef SEGMENTS
580     p.sg:=0
581   }endif
582   end;

584   procedure init1(fsp:sp; fad:integer);
585   begin with s do begin
586     sop:=fsp; packbit:=false; ak:=fixed; pos.ad:=fad; pos.lv:=level;
587   {ifdef SEGMENTS
588     pos.sg:=0;
589   }endif
590   end end;

592   function newip(kl:idclass; n:alpha; idt:tp; nxt:ip):ip;
593   var p:ip; f:iflagset;
594   begin f:=[];
595     case kl of
596       types,varbnd:  {similar structure}
597         new(p,types);
598       konst:
599         begin new(p,konst); p^.value:=0 end;
600       vars:
601         begin new(p,vars); f:=[used,assigned]; initpos(p^.vpos) end;
602       field:
603         begin new(p,field); p^.foffset:=0 end;
604       proc,func:  {same structure}
605         begin new(p,proc,actual); p^.pfkind:=actual;
606           initpos(p^.pfpos); p^.pfno:=0; p^.parhead:=nil; p^.headlc:=0
607         end
608     end;
609     p^.name:=n; p^.klass:=kl; p^.idtype:=idt; p^.next:=nxt;
610     p^.llink:=nil; p^.rlink:=nil; p^.iflag:=f; newip:=p
611   end;

613   function newsp(sf:structform; sz:integer):sp;
614   var p:sp; sflag:sflagset;
615   begin sflag:=[];
616     case sf of
617       scalar:
618         begin new(p,scalar); p^.scalno:=0; p^.fconst:=nil end;
619       subrange:
620         new(p,subrange);
621       pointer:
622         begin new(p,pointer); p^.eltype:=nil end;
623       power:
624         new(p,power);
625       files:
626         begin new(p,files); sflag:=[withfile] end;
627       arrays,carray:  {same structure}
628         new(p,arrays);
629       records:
630         new(p,records);
631       variant:
632         new(p,variant);
633       tag:
634         new(p,tag);
635     end;
636     p^.form:=sf; p^.size:=sz; p^.sflag:=sflag; newsp:=p;
637   end;

639   procedure init1;
640   var c:char;
641   begin
642   {initialise the first name space}
643     new(top,blok); top^.occur:=blok; top^.nlink:=nil; top^.fname:=nil;
644     level:=0;
645   {reserved words}
646     rw[ 0]:='if      ';  rw[ 1]:='do      ';  rw[ 2]:='of      ';
647     rw[ 3]:='to      ';  rw[ 4]:='in      ';  rw[ 5]:='or      ';
648     rw[ 6]:='end     ';  rw[ 7]:='for     ';  rw[ 8]:='nil     ';
649     rw[ 9]:='var     ';  rw[10]:='div     ';  rw[11]:='mod     ';
650     rw[12]:='set     ';  rw[13]:='and     ';  rw[14]:='not     ';
651     rw[15]:='then    ';  rw[16]:='else    ';  rw[17]:='with    ';
652     rw[18]:='case    ';  rw[19]:='type    ';  rw[20]:='goto    ';
653     rw[21]:='file    ';  rw[22]:='begin   ';  rw[23]:='until   ';
654     rw[24]:='while   ';  rw[25]:='array   ';  rw[26]:='const   ';
655     rw[27]:='label   ';  rw[28]:='repeat  ';  rw[29]:='record  ';
656     rw[30]:='downto  ';  rw[31]:='packed  ';  rw[32]:='program ';
657     rw[33]:='function';  rw[34]:='procedur';
658   {corresponding symbols}
659     rsy[ 0]:=ifsy;       rsy[ 1]:=dosy;       rsy[ 2]:=ofsy;
660     rsy[ 3]:=tosy;       rsy[ 4]:=insy;       rsy[ 5]:=orsy;
661     rsy[ 6]:=endsy;      rsy[ 7]:=forsy;      rsy[ 8]:=nilcst;
662     rsy[ 9]:=varsy;      rsy[10]:=divsy;      rsy[11]:=modsy;
663     rsy[12]:=setsy;      rsy[13]:=andsy;      rsy[14]:=notsy;
664     rsy[15]:=thensy;     rsy[16]:=elsesy;     rsy[17]:=withsy;
665     rsy[18]:=casesy;     rsy[19]:=typesy;     rsy[20]:=gotosy;
666     rsy[21]:=filesy;     rsy[22]:=beginsy;    rsy[23]:=untilsy;
667     rsy[24]:=whilesy;    rsy[25]:=arraysy;    rsy[26]:=constsy;
668     rsy[27]:=labelsy;    rsy[28]:=repeatsy;   rsy[29]:=recordsy;
669     rsy[30]:=downtosy;   rsy[31]:=packedsy;   rsy[32]:=progsy;
670     rsy[33]:=funcsy;     rsy[34]:=procsy;
671   {indices into rw to find reserved words fast}
672     frw[0]:= 0; frw[1]:= 0; frw[2]:= 6; frw[3]:=15; frw[4]:=22;
```

```
673       frw[5]:=28; frw[6]:=32; frw[7]:=33; frw[8]:=35;
674     {char types}
675       for c:=chr(0) to chr(maxcharord) do os[c]:=others;
676       for c:='0' to '9' do os[c]:=digit;
677       for c:='A' to 'Z' do os[c]:=upper;
678       for c:='a' to 'z' do os[c]:=lower;
679       os[chr(newline)]:=layout;
680       os[chr(hortab)]:=layout;
681       os[chr(formfeed)]:=layout;
682       os[chr(carret)]:=layout;
683     {characters with corresponding chartype in ASCII order}
684       os[chr(tab)]:=tabch;
685       os[' ']:=layout;      os['"']:=dquotech;      os['''']:=quotech;
686       os['(']:=lparentch;   os[')']:=rparentch;     os['*']:=star;
687       os['+']:=plusch;      os[',']:=commach;       os['-']:=minch;
688       os['.']:=periodch;    os['/']:=slash;         os[':']:=colonch;
689       os[';']:=semich;      os['<']:=lessch;        os['=']:=equal;
690       os['>']:=greaterch;   os['[']:=lbrackch;      os[']']:=rbrackch;
691       os['^']:=arrowch;     os['{']:=lbracech;
692     {single character symbols in chartype order}
693       osy[rparentch]:=rparent;      osy[lbrackch]:=lbrack;
694       osy[rbrackch]:=rbrack;        osy[commach]:=comma;
695       osy[semich]:=semicolon;       osy[arrowch]:=arrow;
696       osy[plusch]:=plussy;          osy[minch]:=minsy;
697       osy[slash]:=slashsy;          osy[star]:=starsy;
698       osy[equal]:=eqsy;
699     end;

700
701     procedure init2;
702     var p,q:ip; k:idclass;
703     begin
704     {undefined identifier pointers used by searchid}
705       for k:=types to func do
706         undefip[k]:=newip(k,spaces,nil,nil);
707     {standard type pointers. some size are filled in by handleopts}
708       intptr   :=newsp(scalar,intsize);
709       realptr  :=newsp(scalar,0);
710       longptr  :=newsp(scalar,longsize);
711       charptr  :=newsp(scalar,charsize);
712       boolptr  :=newsp(scalar,boolsize);
713       nilptr   :=newsp(pointer,0);
714       stringptr:=newsp(pointer,0);
715       emptyset :=newsp(power,intsize); emptyset^.elset:=nil;
716       textptr  :=newsp(files,0); textptr^.filtype:=charptr;
717     {standard type names}
718       enterid(newip(types,'integer ',intptr,nil));
719       enterid(newip(types,'real    ',realptr,nil));
720       enterid(newip(types,'char    ',charptr,nil));
721       enterid(newip(types,'boolean ',boolptr,nil));
722       enterid(newip(types,'text    ',textptr,nil));
723     {standard constant names}
724       q:=nil; p:=newip(konst,'false   ',boolptr,q); enterid(p);
725       q:=p; p:=newip(konst,'true    ',boolptr,q); p^.value:=1; enterid(p);
726       boolptr^.fconst:=p;
727       p:=newip(konst,'maxint  ',intptr,nil); p^.value:=maxint; enterid(p);
728       p:=newip(konst,spaces,charptr,nil); p^.value:=maxcharord;

729       charptr^.fconst:=p;
730     end;

732     procedure init3;
733     var j:standpf; p:ip; q:np;
734         pfn:array[standpf] of alpha;
735         ftype:array[feof..farctan] of sp;
736     begin
737     {names of standard procedures/functions}
738       pfn[pread    ]:='read    ';  pfn[preadln  ]:='readln  ';
739       pfn[pwrite   ]:='write   ';  pfn[pwriteln ]:='writeln ';
740       pfn[pput     ]:='put     ';  pfn[pget     ]:='get     ';
741       pfn[ppage    ]:='page    ';  pfn[preset   ]:='reset   ';
742       pfn[prewrite ]:='rewrite ';  pfn[pnew     ]:='new     ';
743       pfn[pdispose ]:='dispose ';  pfn[ppack    ]:='pack    ';
744       pfn[punpack  ]:='unpack  ';  pfn[pmark    ]:='mark    ';
745       pfn[prelease ]:='release ';  pfn[phalt    ]:='halt    ';
746       pfn[feof     ]:='eof     ';  pfn[feoln     ]:='eoln    ';
747       pfn[fabs     ]:='abs     ';  pfn[fsqr      ]:='sqr     ';
748       pfn[ford     ]:='ord     ';  pfn[fchr      ]:='chr     ';
749       pfn[fpred    ]:='pred    ';  pfn[fsucc     ]:='succ    ';
750       pfn[fodd     ]:='odd     ';  pfn[ftrunc    ]:='trunc   ';
751       pfn[fround   ]:='round   ';  pfn[fsin      ]:='sin     ';
752       pfn[fcos     ]:='cos     ';  pfn[fexp      ]:='exp     ';
753       pfn[fsqrt    ]:='sqrt    ';  pfn[fln       ]:='ln      ';
754       pfn[farctan  ]:='arctan  ';
755     {parameter types of standard functions}
756       ftype[feof   ]:=nil;              ftype[feoln    ]:=nil;
757       ftype[fabs   ]:=nil;              ftype[fsqr     ]:=nil;
758       ftype[ford   ]:=nil;              ftype[fchr     ]:=intptr;
759       ftype[fpred  ]:=nil;              ftype[fsucc    ]:=nil;
760       ftype[fodd   ]:=intptr;           ftype[ftrunc   ]:=nil;
761       ftype[fround ]:=nil;              ftype[fsin     ]:=realptr;
762       ftype[fcos   ]:=realptr;          ftype[fexp     ]:=realptr;
763       ftype[fsqrt  ]:=realptr;          ftype[fln      ]:=realptr;
764       ftype[farctan ]:=realptr;
765     {standard procedure/function identifiers}
766       for j:=pread to phalt do
767         begin new(p,proc,standard); p^.klass:=proc;
768           p^.name:=pfn[j]; p^.pfkind:=standard; p^.key:=j; enterid(p);
769         end;
770       for j:=feof to farctan do
771         begin new(p,func,standard); p^.klass:=func; p^.idtype:=ftype[j];
772           {idtype is used not for result type but for parameter type !! }
773           p^.name:=pfn[j]; p^.pfkind:=standard; p^.key:=j; enterid(p);
774         end;
775     {program identifier}
776       progp:=newip(proc,'_main   ',nil,nil);
777     {new name space for user externals}
778       new(q,blck); q^.occur:=blck; q^.mlink:=top; q^.fname:=nil; top:=q;
779     end;

781     procedure init4;
782     var c:char;
783     begin
784     {pascal library mnemonics}
```



```
785       lmn[ELN ]:='_eln';    lmn[EFL ]:='_efl';      lmn[CLS ]:='_cls';
786       lmn[WDV ]:='_wdv';
787       lmn[OPN ]:='_opn';    lmn[GETX]:='_get';      lmn[RDI ]:='_rdi';
788       lmn[RDC ]:='_rdc';    lmn[RDR ]:='_rdr';      lmn[RDL ]:='_rdl';
789       lmn[RLN ]:='_rln';
790       lmn[CRE ]:='_cre';    lmn[PUTX]:='_put';      lmn[WRI ]:='_wri';
791       lmn[WSI ]:='_wsi';    lmn[WRC ]:='_wrc';      lmn[WSC ]:='_wso';
792       lmn[WRS ]:='_wre';    lmn[WSS ]:='_wss';      lmn[WRB ]:='_wrb';
793       lmn[WSB ]:='_wsb';    lmn[WRR ]:='_wrr';      lmn[WSR ]:='_wsr';
794       lmn[WRL ]:='_wrl';    lmn[WSL ]:='_wsl';
795       lmn[WRF ]:='_wrf';    lmn[WRX ]:='_wrx';      lmn[WSZ ]:='_wsx';
796       lmn[WLN ]:='_wln';    lmn[PAG ]:='_pag';
797       lmn[ABR ]:='_abr';    lmn[RND ]:='_rnd';      lmn[SIN ]:='_sin';
798       lmn[COS ]:='_cos';    lmn[EXPX]:='_exp';      lmn[SQT ]:='_sqt';
799       lmn[LOG ]:='_log';    lmn[ATN ]:='_atn';      lmn[ABI ]:='_abi';
800       lmn[ABL ]:='_abl';
801       lmn[BCP ]:='_bcp';    lmn[BTS ]:='_bts';      lmn[NEWX]:='_new';
802       lmn[SAV ]:='_sav';    lmn[RST ]:='_rst';      lmn[INI ]:='_ini';
803       lmn[HLT ]:='_hlt';    lmn[ASS ]:='_ass';      lmn[GTO ]:='_gto';
804       lmn[PAC ]:='_pac';    lmn[UNP ]:='_unp';      lmn[DIS ]:='_dis';
805       lmn[ASZ ]:='_asz';    lmn[NDX ]:='_ndi';      lmn[MDL ]:='_mdl';
806     {options}
807       for c:='a' to 'z' do begin opt[c]:=0; forceopt[c]:=false end;
808       opt['s']:=on;
809       opt['f']:=floatsize div wordsize;     {default real size in words}
810       opt['i']:=maxsetlist+1;
811       opt['l']:=on;
812       opt['o']:=on;
813       opt['p']:=addrsize div wordsize;      {default pointer size in words}
814       opt['r']:=on;
815       dopt:=off;
816     {smaller variables}
817       b.nesttop:=nil;
818       b.le:=0;
819       b.illmo:=0;
820       b.forwcount:=0;
821       b.lchain:=nil;
822       c.ohno:=0;
823       c.line:=1;
824       c.linr:=1;
825       c.orig:=1;
826       c.fname:=emptyfnam;
827       source:=emptyfnam;
828       line:=0;
829       dlmo:=0;
830       srge:=1;
831       lastpfno:=0;
832       givealine:=true;
833       isecluding:=false;
834       confexpected:=false;
835       intypedcase:=false;
836       fixmsed:=false;
837       secondtok:=false;
838       iop[false]:=nil;
839       iop[true]:=nil;
840       argv[0],ad:=-1;

841       argv[1],ad:=-1;
842     end;

844     procedure handleopts;
845     begin
846       oopt:=opt['o'];
847       dopt:=opt['d'];
848       iopt:=opt['i'];
849       sopt:=opt['s'];
850       realsize:=sopt['f'] * wordsize; realptr^.size:=realsize;
851       ptrsize:=sopt['p'] * wordsize; nilptr^.size:=ptrsize;
852       fhsize:=6*intsize + 2*ptrsize;
853       textptr^.size:=fhsize+bufsize; stringptr^.size:=ptrsize;
854       if sopt<>off then begin oopt:=off; dopt:=off end
855       else if opt['u']<>off then os['_']:=lower;
856       if oopt<>off then enterid(newip(types,'string  ',stringptr,nil));
857       if dopt<>off then enterid(newip(types,'long    ',longptr,nil));
858       if opt['o']=off then begin gen1(ps_mes,mesoptoff); genend end;
859       if ptrsize<>wordsize then begin gen1(ps_mes,mesvirtual); genend end;
860       if dopt<>off then fltused:=true;  {temporary kludge}
861     end;

863     {=============================================================}

865     procedure trace(tname:alpha; fip:ip; var namdlb:integer);
866     var i:integer;
867     begin
868       if opt['t']<>off then
869         begin
870           if namdlb=0 then
871             begin dlbno:=dlbno+1; namdlb:=dlbno; gendlb(dlbno);
872               gen0(ps_run); write(eml,sp_scon,8);
873               for i:=1 to 8 do write(eml,ord(fip^.name[i]));  genend;
874             end;
875           gen1(op_ujp,9); gen0(op_lao,namdlb);
876           gen0(op_cal); gen1dnt(sp_pnam,tname);
877         end;
878     end;

880     function forwof(fsp:sp; forus:forwset):boolean;
881     begin if fsp=nil then forwof:=false else forwof:=fsp^.form in forus end;

883     function sizeof(fsp:sp):integer;
884     var s:integer;
885     begin s:=0;
886       if fsp<>nil then s:=fsp^.size;
887       if s<>1 then if odd(s) then s:=s+1;
888       sizeof:=s
889     end;

891     function even(i:integer):integer;
892     begin if odd(i) then i:=i+1; even:=i end;

894     procedure exchange(l1,l2:integer);
895     var d1,d2:integer;
896     begin d1:=l2-l1; d2:=limo-l2;
```

```
897       if (d1<>0) and (d2<>0) then
898         begin gen1(pa_exc,d1); genost(d2) end
899     end;

901     procedure setop(m:byte);
902     begin gen1(m,even(sizeof(s.asp))) end;

904     procedure expandemptyset(fsp:sp);
905     var i:integer;
906     begin
907       for i:=2 to sizeof(fsp) div wordsize do gen1(op_loc,0); s.asp:=fsp
908     end;

910     procedure push(local:boolean; sd:integer; sz:integer);
911     begin assert not odd(sz);
912       if sz>wordsize then
913         begin if local then gen1(op_lal,sd) else gen1(op_lae,sd);
914           gen1(op_loi,sz)
915         end
916       else
917         if local then gen1(op_lol,sd) else gen1(op_loe,sd)
918     end;

920     procedure pop(local:boolean; sd:integer; sz:integer);
921     begin assert not odd(sz);
922       if sz>wordsize then
923         begin if local then gen1(op_lal,sd) else gen1(op_lae,sd);
924           gen1(op_sti,sz)
925         end
926       else
927         if local then gen1(op_stl,sd) else gen1(op_ste,sd)
928     end;

930     procedure lexical(m:byte; lv:integer; sd:integer; sz:integer);
931     begin gen1(op_lex,level-lv); gen1(op_adi,sd); gen1(m,sz) end;

933     procedure loadpos(var p:position; sz:integer);
934     begin with p do
935       if lv<>0 then
936     #ifdef SEGMENTS
937         if sg<>0 then
938           begin gen1(op_lsa,sg); gen1(op_adi,sd); gen1(op_loi,sz) end
939         else
940     #endif
941           push(global,sd,sz)
942       else
943         if lv=level then push(local,sd,sz) else
944           lexical(op_loi,lv,sd,sz);
945     end;

947     procedure descraddr(var p:position);
948     begin if p.lv=0 then gen0(op_lae,p.sd) else loadpos(p,ptrsize) end;

950     procedure loadaddr;
951     begin with s do begin
952       case ak of
953         fixed:
954           with pos do
955             if lv<>0 then
956     #ifdef SEGMENTS
957               if sg<>0 then
958                 begin gen1(op_lsa,sg); gen1(op_adi,sd) end
959               else
960     #endif
961                 gen1(op_lae,sd)
962             else
963               if lv=level then gen1(op_lal,sd) else
964                 begin gen1(op_lex,level-lv); gen1(op_adi,sd) end;
965         pfixed:
966           loadpos(pos,ptrsize);
967         ploaded:
968           ;
969         indexed:
970           gen0(op_sas);
971       end;  {case}
972       ak:=ploaded;
973     end end;

975     procedure load;
976     var sz:integer;
977     begin with s do begin
978       sz:=sizeof(asp); if not packbit then sz:=even(sz);
979       if asp<>nil then
980         case ak of
981           cst:
982             gen1(op_loc,pos.sd);  {only one-word scalars}
983           fixed:
984             loadpos(pos,sz);
985           pfixed:
986             begin loadpos(pos,ptrsize); gen1(op_loi,sz) end;
987           loaded:
988             ;
989           ploaded:
990             gen1(op_loi,sz);
991           indexed:
992             gen0(op_las);
993         end;  {case}
994       ak:=loaded;
995     end end;

997     procedure store;
998     var sz:integer;
999     begin with s do begin
1000      sz:=sizeof(asp); if not packbit then sz:=even(sz);
1001      if asp<>nil then
1002        case ak of
1003          fixed:
1004            with pos do
1005              if lv<>0 then
1006    #ifdef SEGMENTS
1007                if sg<>0 then
1008                  begin gen1(op_lsa,sg);
```



```
1009                gen1(op_adi,sd); gen1(op_sti,sz)
1010              end
1011            else
1012    #endif
1013              pop(global,sd,sz)
1014          else
1015            if level=lv then pop(local,sd,sz) else
1016              lexical(op_sti,lv,sd,sz);
1017        pfixed:
1018          begin loadpos(pos,ptrsize); gen1(op_sti,sz) end;
1019        ploaded:
1020          gen1(op_sti,sz);
1021        indexed:
1022          gen0(op_sas);
1023      end;  {case}
1024    end end;

1026    procedure fieldaddr(off:integer);
1027    begin with s do
1028      if (ak=fixed) and not packbit then pos.sd:=pos.sd+off else
1029        begin loadaddr; gen1(op_adi,off) end
1030    end;

1032    procedure loadcheap;
1033    begin if formof(s.asp,[arrays..records]) then loadaddr else load end;

1035    {*****************************************************************}

1037    procedure nextch;
1038    begin
1039      eol:=eoln(input); read(input,ch); s.chno:=s.chno+1; chsy:=sss[ch];
1040    end;

1042    procedure nextln;
1043    begin
1044      if eof(input) then
1045        begin
1046          if not eofexpected then error(403) else
1047          begin
1048            if fltused then begin gen1(ps_mes,mesfloats); genend end;
1049            gen0(ps_eof)
1050          end;
1051    #ifdef STANDARD
1052          goto 9999
1053    #endif
1054    #ifndef STANDARD
1055          halt
1056    #endif
1057        end;
1058      s.chno:=0; s.lino:=s.lino+1; s.liar:=s.liar+1;
1059      if not ineluding then
1060        begin s.orig:=s.orig+1; givaline:=true end;
1061    end;

1063    procedure options(normal:boolean);
1064    var o,ci:char; i:integer;

1066    procedure getc;
1067    var b:byte;
1068    begin
1069      if normal then
1070        begin nextch; o:=ch end
1071      else
1072        begin read(em1,b); o:=chr(b) end
1073    end;

1075    begin
1076      repeat getc;
1077        if (o>='a') and (o<='z') then
1078          begin ci:=o; getc; i:=0;
1079            if o='+' then begin i:=1; getc end else
1080            if o='-' then getc else
1081            if o in[o]+digit then
1082              repeat i:=i*10 + ord(o) - ord('0'); getc;
1083              until o in[o]<>digit
1084            else i:=-1;
1085            if i>=0 then
1086              if not normal then
1087                begin forceopt[ci]:=true; opt[ci]:=i end
1088              else
1089                if not forceopt[ci] then opt[ci]:=i;
1090          end;
1091      until o<>',';
1092    end;

1094    procedure linedirective;
1095    var i,j:integer;
1096    begin i:=0; j:=0;
1097      repeat nextch until (ch<>' ') or eol;
1098      while chsy=digit do
1099        begin i:=i*10 + ord(ch) - ord('0'); nextch end;
1100      while (ch=' ') and not eol do nextch;
1101      if (ch<>'"') or (i=0) then error(404) else
1102        begin nextch;
1103          while (ch<>'"') and not eol do
1104            begin
1105              if ch='/' then j:=0 else
1106                begin if j<0 then s.fnam:=emptyfnam;
1107                  j:=j+1; if j<=fnmax then s.fnam[j]:=ch;
1108                end;
1109              nextch
1110            end;
1111          if source=emptyfnam then source:=s.fnam;
1112          inoluding:=source<>s.fnam;
1113          i:=i-1; s.liar:=i;
1114          if not inoluding then s.orig:=i
1115        end;
1116      while not eol do nextch;
1117    end;

1119    procedure putdig;
1120    begin ix:=ix+1; if ix<=rmax then strbuf[ix]:=ch; nextch end;
```

```
1122    procedure inident;
1123    label 1;
1124    var i,k:integer;
1125    begin k:=0; id:=spaces;
1126      repeat
1127        if ch=upper then ch:=chr(ord(ch)-ord('A')+ord('a'));
1128        if k<idmax then begin k:=k+1; id[k]:=ch end;
1129        nextch
1130      until ch<>digit;
1131                 {lower=0,upper=1,digit=2. ugly but fast}
1132      for i:=frw[k-1] to frw[k] - 1 do
1133        if rw[i]=id then
1134          begin sy:=rsy[i]; goto 1 end;
1135      sy:=ident;
1136    1:
1137    end;

1139    procedure innumber;
1140    label   1;
1141    const   imax = 10;
1142    var     i:integer;
1143            is:packed array[1..imax] of char;
1144    begin ix:=0; sy:=intcst; val:=0;
1145      repeat putdig until ch<>digit;
1146      if (ch='.') or (ch='e') or (ch='E') then
1147        begin
1148          if ch='.' then
1149            begin putdig;
1150              if ch='.' then
1151                begin secondd0t:=true; ix:=ix-1; goto 1 end;
1152              if ch<>digit then error(+05) else
1153                repeat putdig until ch<>digit;
1154            end;
1155          if (ch='e') or (ch='E') then
1156            begin putdig;
1157              if (ch='+') or (ch='-') then putdig;
1158              if ch<>digit then error(+06) else
1159                repeat putdig until ch<>digit;
1160            end;
1161          if ix>rmax then begin error(+07); ix:=rmax end;
1162          sy:=realcst; fltused:=true; dlbno:=dlbno+1; val:=dlbno;
1163          gendlb(dlbno); gen0(pa_rom); write(em1,sp_room,ix);
1164          for i:=1 to ix do write(em1,ord(strbuf[i])); genend;
1165        end;
1166    1:if (ch='x') or (ch='X') then teststandard;
1167      if sy=intcst then
1168        if ix>imax then error(+08) else
1169          begin ix:='0000000000'; i:=imax+1;
1170            while ix>0 do
1171              begin i:=i-1; is[i]:=strbuf[ix]; ix:=ix-1 end;
1172            if is<=maxintstring then
1173              while i<=imax do
1174                begin val:=val*10 - ord('0') + ord(is[i]); i:=i+1 end
1175            else if (is<=maxlongstring) and (dopt<>off) then
1176              begin sy:=longcst; dlbno:=dlbno+1; val:=dlbno;
1177                gendlb(dlbno); gen0(pa_con); write(em1,sp_loon,imax+1-i);
1178                while i<=imax do
1179                  begin write(em1,ord(is[i])); i:=i+1 end;
1180                genend
1181              end
1182            else error(+09)
1183          end
1184    end;

1186    procedure instring(qc:char);
1187    var i:integer;
1188    begin ix:=0; zerostring:=qc='';
1189      repeat
1190        repeat nextch; ix:=ix+1; if ix<=smax then strbuf[ix]:=ch;
1191          until (ch=qc) or eol;
1192        if ch=qc then nextch else error(+010);
1193      until ch<>qc;
1194      if not zerostring then
1195        begin ix:=ix-1; if ix=0 then error(+011) end
1196      else
1197        begin strbuf[ix]:=chr(0); if opt=off then error(+012) end;
1198      if (ix=1) and not zerostring then
1199        begin sy:=charcst; val:=ord(strbuf[1]) end
1200      else
1201        begin sy:=stringcst; dlbno:=dlbno+1; val:=dlbno;
1202          if ix>smax then begin error(+013); ix:=smax end;
1203          gendlb(dlbno); gen0(pa_rom); write(em1,sp_scon,ix);
1204          for i:=1 to ix do write(em1,ord(strbuf[i])); genend;
1205        end
1206    end;

1208    procedure incomment;
1209    var stopc:char;
1210    begin nextch; stopc:=':';
1211      if ch='$' then options(true);
1212      while (ch<>'}') and (ch<>stopc) do
1213        begin stopc:='}'; if ch='"' then stopc:='"';
1214          if ch=';' then error(-(+018));
1215          if eol then nextln; nextch
1216        end;
1217      if ch<>'}' then teststandard;
1218      nextch
1219    end;

1221    procedure insym;
1222      {read next basic symbol of source program and return its
1223      description in the global variables sy, op, id, val and ix}
1224    label 1;
1225    begin
1226    1:case chsy of
1227        tabch:
1228          begin c.chno:=c.chno - c.chno mod 8 + 8; nextch; goto 1 end;
1229        layout:
1230          begin if eol then nextln; nextch; goto 1 end;
1231        lower,upper: inident;
1232        digit: innumber;
```



```
1233        quotech,dquotech:
1234          instring(ch);
1235        colonch:
1236          begin nextch;
1237            if ch='=' then begin sy:=becomes; nextch end else sy:=colon;
1238            end;
1239        periodch:
1240          begin nextch;
1241            if secondd0t then begin secondd0t:=false; sy:=colon2 end else
1242            if ch='.' then begin sy:=colon2; nextch end else sy:=period
1243            end;
1244        lessch:
1245          begin nextch;
1246            if ch='=' then begin sy:=leqy; nextch end else
1247            if ch='>' then begin sy:=neqy; nextch end else sy:=ltgy
1248            end;
1249        greaterch:
1250          begin nextch;
1251            if ch='=' then begin sy:=geqy; nextch end else sy:=gtgy
1252            end;
1253        lparentch:
1254          begin nextch;
1255            if ch<>'*' then sy:=lparent else
1256              begin teststandard; incomment; goto 1 end;
1257            end;
1258        lbracech:
1259          begin incomment; goto 1 end;
1260        rparentch,lbrackch,rbrackch,commach,semich,arrowch,
1261        plusch,minch,slash,star,equal:
1262          begin sy:=sxy[chsy]; nextch end;
1263        others:
1264          begin
1265            if (ch='$') and (c.chno=1) then linedirective else
1266              begin error(+015); nextch end;
1267            goto 1
1268          end;
1269      end {case}
1270    end;

1272    procedure nextif(fsy:symbol; err:integer);
1273    begin if sy=fsy then insym else error(-err) end;

1275    function find1(sys1,sys2:sos; err:integer):boolean;
1276    {symbol of sys1 expected. return true if sy in sys1}
1277    begin
1278      if not (sy in sys1) then
1279        begin error(err); while not (sy in sys1+sys2) do insym end;
1280      find1:=sy in sys1
1281    end;

1283    function find2(sys1,sys2:sos; err:integer):boolean;
1284    {symbol of sys1+sys2 expected. return true if sy in sys1}
1285    begin
1286      if not (sy in sys1+sys2) then
1287        begin error(err); repeat insym until sy in sys1+sys2 end;
1288      find2:=sy in sys1
```

```
1289    end;

1291    function find3(sy1:symbol; sys2:sos; err:integer):boolean;
1292    {symbol sy1 or one of sys2 expected. return true if sy1 found and skip}
1293    begin find3:=true;
1294      if not (sy in [sy1]+sys2) then
1295        begin error(err); repeat insym until sy in [sy1]+sys2 end;
1296      if sy=sy1 then insym else find3:=false
1297    end;

1299    function endofloop(sys1,sys2:sos; sy3:symbol; err:integer):boolean;
1300    begin endofloop:=false;
1301      if find2(sys2+[sy3],sys1,err) then nextif(sy3,err+1)
1302      else endofloop:=true;
1303    end;

1305    function lastsemicolon(sys1,sys2:sos; err:integer):boolean;
1306    begin lastsemicolon:=true;
1307      if not endofloop(sys1,sys2,semicolon,err) then
1308        if find2(sys2,sys1,err+2) then lastsemicolon:=false
1309    end;

1311    {*************************************************************}

1313    function searchid(fids1a: setofids):ip;
1314    {search for current identifier symbol in the name table}
1315    label 1;
1316    var lip:ip; ic:idclass;
1317    begin lastsp:=top;
1318      while lastsp<>nil do
1319        begin lip:=lastsp^.fname;
1320          while lip<>nil do
1321            if lip^.name=id then
1322              if lip^.klass in fids1 then
1323                begin
1324                  if lip^.klass=vars then if lip^.vpos.lv<>level then
1325                    lip^.iflag:=lip^.iflag+[moreq];
1326                  goto 1
1327                end
1328              else lip:=lip^.rlink
1329            else
1330              if lip^.name< id then lip:=lip^.rlink else lip:=lip^.llink;
1331          lastsp:=lastsp^.nlink;
1332        end;
1333      errid(+016,id);
1334      if types in fids1 then ic:=types else
1335      if vars  in fids1 then ic:=vars else
1336      if konst in fids1 then ic:=konst else
1337      if proc  in fids1 then ic:=proc else
1338      if func  in fids1 then ic:=func else ic:=field;
1339      lip:=undefip[ic];
1340    1:
1341      searchid:=lip
1342    end;

1344    function searchsection(fip: ip):ip;
```

```
1345    {to find record fields and forward declared procedure id's
1346      -->procedure pfdeclaration
1347      -->procedure selector}
1348    label 1;
1349    begin
1350      while fip<>nil do
1351        if fip^.name=id then goto 1 else
1352          if fip^.name< id then fip:=fip^.rlink else fip:=fip^.llink;
1353    1:  searchsection:=fip
1354    end;

1356    function searchlab(flp:lp; val:integer):lp;
1357    label 1;
1358    begin
1359      while flp<>nil do
1360        if flp^.labval=val then goto 1 else flp:=flp^.nextlp;
1361    1:searchlab:=flp
1362    end;

1364    procedure opconvert(ts:twostruct);
1365    var op:integer;
1366    begin with s do begin
1367      case ts of
1368        ir: begin op:=op_oif; asp:=realptr; fltused:=true end;
1369        ri: begin op:=op_ofi; asp:=intptr; fltused:=true end;
1370        il: begin op:=op_oid; asp:=longptr end;
1371        li: begin op:=op_odi; asp:=intptr end;
1372        lr: begin op:=op_odf; asp:=realptr; fltused:=true end;
1373        rl: begin op:=op_ofd; asp:=longptr; fltused:=true end;
1374      end;
1375      gen0(op)
1376    end end;

1378    procedure negate(l1:integer);
1379    var l2:integer;
1380    begin
1381      if s.asp=intptr then gen0(op_neg) else
1382        begin l2:=line; gen1(op_loc,0);
1383          if s.asp=longptr then
1384            begin opconvert(il); exchange(l1,l2); gen0(op_dab) end
1385          else  {realptr}
1386            begin opconvert(ir); exchange(l1,l2); gen0(op_fsb) end
1387        end
1388    end;

1390    function desub(fsp:sp):sp;
1391    begin
1392      if formof(fsp,[subrange]) then fsp:=fsp^.rangetype; desub:=fsp
1393    end;

1395    function nicescalar(fsp:sp):boolean;
1396    begin
1397      if fsp=nil then nicescalar:=true else
1398        nicescalar:=(fsp^.form=scalar) and (fsp<>realptr) and (fsp<>longptr)
1399    end;

1401    function bounds(fsp:sp; var fmin,fmax:integer):boolean;
1402    {compute bounds if possible, else return false}
1403    begin bounds:=false; fmin:=0; fmax:=0;
1404      if fsp<>nil then
1405        if fsp^.form=subrange then
1406          begin fmin:=fsp^.min; fmax:=fsp^.max; bounds:=true end else
1407        if fsp^.form=scalar then
1408          if fsp^.fconst<>nil then
1409            begin fmin:=0; fmax:=fsp^.fconst^.value; bounds:=true end
1410    end;

1412    procedure genrok(fsp:sp);
1413    var min,max,sno:integer;
1414    begin
1415      if opt['r']<>off then if bounds(fsp,min,max) then
1416        begin
1417          if fsp^.form=scalar then sno:=fsp^.scalno else sno:=fsp^.subrno;
1418          if sno=0 then
1419            begin dlbno:=dlbno+1; sno:=dlbno;
1420              gendlb(dlbno); gen1(ps_rom,min); genost(max); genend;
1421              if fsp^.form=scalar then fsp^.scalno:=sno else
1422                fsp^.subrno:=sno
1423            end;
1424          gend(op_rok,sno);
1425        end
1426    end;

1428    procedure checkbnds(fsp:sp);
1429    var min1,max1,min2,max2:integer; bool:boolean;
1430    begin
1431      if bounds(fsp,min1,max1) then
1432        begin bool:=bounds(s.asp,min2,max2);
1433          if (bool=false) or (min2<min1) or (max2>max1) then
1434            genrok(fsp);
1435        end;
1436      s.asp:=fsp;
1437    end;

1439    function eqstruct(p,q:sp):boolean;
1440    begin eqstruct:=(p=q) or (p=nil) or (q=nil) end;

1442    function string(fsp:sp):boolean;
1443    var lsp:sp;
1444    begin string:=false;
1445      if formof(fsp,[arrays]) then
1446        if eqstruct(fsp^.aeltype,charptr) then
1447          if spack in fsp^.sflag then
1448            begin lsp:=fsp^.inxtype;
1449              if lsp=nil then string:=true else
1450                if lsp^.form=subrange then
1451                  if lsp^.rangetype=intptr then
1452                    if lsp^.min=1 then
1453                      string:=true
1454            end
1455    end;
```



```
1457    function compat(p,q:sp):twostruct;
1458    begin compat:=noteq;
1459      if eqstruct(p,q) then compat:=eq else
1460        begin p:=desub(p); q:=desub(q);
1461          if eqstruct(p,q) then compat:=subeq else
1462          if p^.form=q^.form then
1463            case p^.form of
1464              scalar:
1465                if (p=intptr) and (q=realptr) then compat:=ir else
1466                if (p=realptr) and (q=intptr) then compat:=ri else
1467                if (p=intptr) and (q=longptr) then compat:=il else
1468                if (p=longptr) and (q=intptr) then compat:=li else
1469                if (p=longptr) and (q=realptr) then compat:=lr else
1470                if (p=realptr) and (q=longptr) then compat:=rl else
1471                  ;
1472              pointer:
1473                if (p=nilptr) or (q=nilptr) then compat:=eq;
1474              power:
1475                if p=emptyset then compat:=es else
1476                if q=emptyset then compat:=se else
1477                if ocompat(p^.elset,q^.elset) <= subeq then
1478                  if p^.sflag=q^.sflag then compat:=eq;
1479              arrays:
1480                if string(p) and string(q) and (p^.size=q^.size) then
1481                  compat:=eq;
1482              files,carray,records: ;
1483            end;
1484        end
1485    end;

1487    procedure checkasp(fsp:sp; err:integer);
1488    var ts:twostruct;
1489    begin
1490      ts:=compat(s.asp,fsp);
1491      case ts of
1492        eq:
1493          if fsp<>nil then if withfile in fsp^.sflag then asperr(err);
1494        subeq:
1495          checkbnds(fsp);
1496        li:
1497          begin opconvert(ts); checkasp(fsp,err) end;
1498        il,rl,lr,ir:
1499          opconvert(ts);
1500        es:
1501          expandemptyset(fsp);
1502        noteq,ri,se:
1503          asperr(err);
1504      end
1505    end;

1507    procedure force(fsp:sp; err:integer);
1508    begin load; checkasp(fsp,err) end;

1510    function newident(kl:idclass; idt:sp; nxt:ip; err:integer):ip;
1511    begin newident:=nil;
1512      if sy<>ident then error(err) else
1513        begin newident:=newip(kl,id,idt,nxt); insym end
1514    end;

1516    function stringstruct:sp;
1517    var lsp:sp;
1518    begin {only used when ix and zerostring are still valid}
1519      if zerostring then lsp:=stringptr else
1520        begin lsp:=newsp(arrays,ix*charsize); lsp^.sflag:=[spack];
1521          lsp^.aeltype:=charptr; lsp^.inxtype:=nil;
1522        end;
1523      stringstruct:=lsp;
1524    end;

1526    function address(var lc:integer; sz:integer; pack:boolean):integer;
1527    begin
1528      if lc >= maxint-sz then begin error(+017); lc:=0 end;
1529      if (not pack) or (sz>1) then if odd(lc) then lc:=lc+1;
1530      address:=lc;
1531      lc:=lc+sz
1532    end;

1534    function reserve(s:integer):integer;
1535    var r:integer;
1536    begin r:=address(b.lc,s,false); genreg(r,s,100); reserve:=r;
1537      if b.lc>lcmax then lcmax:=b.lc
1538    end;

1540    function arraysize(fsp:sp; pack:boolean):integer;
1541    var es,min,max,tot,n:integer;
1542    begin es:=sizeof(fsp^.aeltype);
1543      if not pack then es:=wrem(es);
1544      if bounds(fsp^.inxtype,min,max) then;  {we checked before}
1545      dlbno:=dlbno+1; fsp^.arpos.lv:=0; fsp^.arpos.ad:=dlbno;
1546      gendlb(dlbno); gen1(ps_rom,min); genost(max-min);
1547      genost(es); genend;
1548      n:=max-min+1; tot:=es*n;
1549      if es<>0 then if tot div es <> n then begin error(+018); tot:=0 end;
1550      arraysize:=tot
1551    end;

1553    procedure treewalk(fip:ip);
1554    var lsp:sp; i:integer;
1555    begin
1556      if fip<>nil then
1557        begin treewalk(fip^.llink); treewalk(fip^.rlink);
1558          if fip^.klass=vars then
1559            begin if not (used in fip^.iflag) then errid(-(+019),fip^.name);
1560              if not (assigned in fip^.iflag) then errid(-(+020),fip^.name);
1561              lsp:=fip^.idtype;
1562              if not (noreg in fip^.iflag) then
1563                genreg(fip^.vpos.ad,sizeof(lsp),ord(formof(lsp,[pointer])));
1564              if lsp<>nil then if withfile in lsp^.sflag then
1565                if lsp^.form=files then
1566                  if level=1 then
1567                    begin
1568                      for i:=2 to argc do with argv[i] do
```

```
1569                    if name=fip^.name then ad:=fip^.vpos.ad
1570                  end
1571                else
1572                  begin
1573                    if not (refer in fip^.iflag) then
1574                      begin gen1(op_mrk,0);
1575                        gen1(op_lal,fip^.vpos.ad); genxp(CLS)
1576                      end
1577                  end
1578              else
1579                if level<>1 then errid(-(+021),fip^.name)
1580            end
1581        end
1582    end;

1584    procedure constant(fsys:sos; var fsp:sp; var fval:integer);
1585    var signed,min:boolean; lip:ip;
1586    begin signed:=(sy=plussy) or (sy=minsy);
1587      if signed then begin min:=sy=minsy; insym end else min:=false;
1588      if find1([ident..nilcst],fsys,+022) then
1589        begin fval:=val;
1590          case sy of
1591            stringcst: fsp:=stringstruct;
1592            charcst: fsp:=charptr;
1593            intcst: fsp:=intptr;
1594            realcst: fsp:=realptr;
1595            longcst: fsp:=longptr;
1596            nilcst: fsp:=nilptr;
1597            ident:
1598              begin lip:=searchid([konst]);
1599                fsp:=lip^.idtype; fval:=lip^.value;
1600              end
1601          end;  {case}
1602          if signed then
1603            if (fsp<>intptr) and (fsp<>realptr) and (fsp<>longptr) then
1604              error(+023)
1605            else if min then fval:= -fval;
1606                      {note: negating the v-number for reals and longs}
1607          insym;
1608        end
1609      else begin fsp:=nil; fval:=0 end;
1610    end;

1612    function cstinteger(fsys:sos; fsp:sp; err:integer):integer;
1613    var lsp:sp; lval,min,max:integer;
1614    begin constant(fsys,lsp,lval);
1615      if fsp<>lsp then
1616        if comptypes(desub(fsp),lsp) then
1617          begin
1618            if boundds(fsp,min,max) then
1619              if (lval<min) or (lval>max) then error(+024)
1620          end
1621        else
1622          begin error(err); lval:=0 end;
1623      cstinteger:=lval
1624    end;

1626    {*****************************************************************}

1628    function typid(err:integer):sp;
1629    var lip:ip; lsp:sp;
1630    begin lsp:=nil;
1631      if sy<>ident then error(err) else
1632        begin lip:=searchid([types]); lsp:=lip^.idtype; insym end;
1633      typid:=lsp
1634    end;

1636    function simpletyp(fsys:sos):sp;
1637    var lsp,lsp1:sp; lip,hip:ip; min,max:integer; lnp:np;
1638        newsubrange:boolean;
1639    begin lsp:=nil;
1640      if find1([ident..lparent],fsys,+025) then
1641        if sy=lparent then
1642          begin insym; lnp:=top;   {decl. consts local to innermost block}
1643            while top^.occur<>blck do top:=top^.slink;
1644            lsp:=newsp(scalar,wordsize); hip:=nil; max:=0;
1645            repeat lip:=newident(konst,lsp,hip,+026);
1646              if lip<>nil then
1647                begin enterid(lip);
1648                  hip:=lip; lip^.value:=max; max:=max+1
1649                end;
1650            until endofloop(fsys+[rparent],[ident],comma,+027);  {+028}
1651            if max<t8 then lsp^.size:=bytesize;
1652            lsp^.fconst:=hip; top:=lnp; nextif(rparent,+029);
1653          end
1654        else
1655          begin newsubrange:=true;
1656            if sy=ident then
1657              begin lip:=searchid([types,konst]); insym;
1658                if lip^.klass=types then
1659                  begin lsp:=lip^.idtype; newsubrange:=false end
1660                else
1661                  begin lsp1:=lip^.idtype; min:=lip^.value end
1662              end
1663            else constant(fsys+[colon2,ident..plussy],lsp1,min);
1664            if newsubrange then
1665              begin lsp:=newsp(subrange,wordsize); lsp^.subrno:=0;
1666                if not scalar(lsp1) then
1667                  begin error(+030); lsp1:=nil; min:=0 end;
1668                lsp^.rangetype:=lsp1;
1669                nextif(colon2,+031); max:=cstinteger(fsys,lsp1,+032);
1670                if min>max then begin error(+033); max:=min end;
1671                if (min>=0) and (max<t8) then lsp^.size:=bytesize;
1672                lsp^.min:=min; lsp^.max:=max
1673              end
1674          end;
1675      simpletyp:=lsp
1676    end;

1678    function arraytyp(fsys:sos;
1679                      artyp:structform;
1680                      aflag:aflagset;
```

```
1681                    function element(fsys:sos):sp
1682                    ):sp;
1683    var lsp,lsp1,hsp:sp; min,max:integer; ok:boolean; sepsy:symbol; lip:ip;
1684        oksys:sos;
1685    begin insym; nextif(lbrack,+034); hsp:=nil;
1686      repeat lsp:=newsp(artyp,0); initpos(lsp^.apos);
1687        lsp^.eltype:=hsp; hsp:=lsp;  {link reversed}
1688        if artyp=carray then
1689          begin sepsy:=semicolon; oksys:=[ident];
1690            lip:=newident(varrbnd,lsp,nil,+035);
1691            if lip<>nil then enterid(lip);
1692            nextif(colon2,+036);
1693            lip:=newident(varrbnd,lsp,lip,+037);
1694            if lip<>nil then enterid(lip);
1695            nextif(colon1,+038); lsp1:=typid(+039);
1696            ok:=aiocscalar(desub(lsp1));
1697          end
1698        else
1699          begin sepsy:=comma; oksys:=[ident..lparent];
1700            lsp1:=simpletyp(fsys+[comma,rbrack,ofsy,ident..packedsy]);
1701            ok:=bounds(lsp1,min,max)
1702          end;
1703        if not ok then begin error(+040); lsp1:=nil end;
1704        lsp^.inxtype:=lsp1
1705      until endofloop(fsys+[rbrack,ofsy,ident..packedsy],oksys,
1706                      sepsy,+041);  {+042}
1707      nextif(rbrack,+043); nextif(ofsy,+044);
1708      lsp:=element(fsys);
1709      if lsp<>nil then aflag:=aflag + lsp^.aflag * [withfile];
1710      repeat  {reverse links and compute size}
1711        hsp:=lsp^.eltype; lsp^.eltype:=hsp; hsp^.aflag:=aflag;
1712        if artyp=arrays then lsp^.size:=arraysize(hsp,speck in aflag);
1713        lsp:=hsp; hsp:=lsp1
1714      until hsp=nil;  {lsp points to array with highest dimension}
1715      arraytyp:=lsp
1716    end;

1718    function typ(fsys:sos):sp;
1719    var lsp,lsp1:sp; oc,mc,min,max:integer;
1720        aflag:aflagset; lnp:np;

1722    function fldlist(fsys:sos):sp;
1723          {level 2: <<  typ}
1724    var fip,hip,lip:ip; lsp:sp;

1726    function varpart(fsys:sos):sp;
1727          {level 3: <<  fldlist <<  typ}
1728    var tip,lip:ip; lvp,headsp,hsp,vsp,tsp,tsp1,tfsp:sp;
1729        minoc,memoc,int,nvar:integer; lid:alpha;
1730    begin insym; tip:=nil; lip:=nil;
1731      tsp:=newsp(tag,0);
1732      if sy<>ident then error(+045) else
1733        begin lid:=id; insym;
1734          if sy=colon1 then
1735            begin tip:=enterip(field,lid,nil,nil); enterid(tip); insym;
1736              if sy<>ident then error(+046) else
```

```
1737                begin lid:=id; insym end;
1738            end;
1739          if sy=ofsy then  {otherwise you may destroy id}
1740            begin id:=lid; lip:=searchid([types]) end;
1741        end;
1742      if lip=nil then tfsp:=nil else tfsp:=lip^.idtype;
1743      if bounds(tfsp,int,nvar) then nvar:=nvar-int+1 else
1744        begin nvar:=0;
1745          if tfsp<>nil then begin error(+047); tfsp:=nil end
1746        end;
1747      tsp^.tfldsp:=tfsp;
1748      if tip<>nil then  {explicit tag}
1749        begin tip^.idtype:=tfsp;
1750          tip^.foffset:=address(oc,sizeof(tfsp),speck in aflag)
1751        end;
1752      nextif(ofsy,+048); minoc:=oc; maxoc:=minoc; headsp:=nil;
1753      repeat hsp:=nil;  {for each caselabel list}
1754        repeat nvar:=nvar-1;
1755          int:=cstinteger(fsys+[ident..plussy,comma,colon1,lparent,
1756                                semicolon,casesy,rparent],tfsp,+049);
1757          lsp:=headsp;  {each label may occur only once}
1758          while lsp<>nil do
1759            begin if lsp^.varval=int then error(+050);
1760              lsp:=lsp^.nxtvar
1761            end;
1762          vsp:=newsp(variant,0); vsp^.varval:=int;
1763          vsp^.nxtvar:=headsp; headsp:=vsp;  {chain of case labels}
1764          vsp^.subtap:=hsp; hsp:=vsp;
1765                {use this field to link labels with same variant}
1766        until endofloop(fsys+[colon1,lparent,semicolon,casesy,rparent],
1767                        [ident..plussy],comma,+051);  {+052}
1768        nextif(colon1,+053); nextif(lparent,+054);
1769        tsp1:=fldlist(fsys+[rparent,semicolon,ident..plussy]);
1770        if oc>maxoc then maxoc:=oc;
1771        while vsp<>nil do
1772          begin vsp^.size:=oc; hsp:=vsp^.subtap;
1773            vsp^.subtap:=tsp1; vsp:=hsp
1774          end;
1775        nextif(rparent,+055);
1776        oc:=minoc;
1777      until lastsemicolon(fsys,[ident..plussy],+056);  {+057 +058}
1778      if nvar>0 then error(-(+059));
1779      tsp^.fstvar:=headsp; tsp^.size:=minoc; oc:=maxoc; varpart:=tsp;
1780    end;

1782    begin  {fldlist}
1783      if find2([ident],fsys+[casesy],+060) then
1784        repeat lip:=nil; hip:=nil;
1785          repeat fip:=newident(field,nil,nil,+061);
1786            if fip<>nil then
1787              begin enterid(fip);
1788                if lip=nil then hip:=fip else lip^.next:=fip; lip:=fip;
1789              end;
1790          until endofloop(fsys+[colon1,ident..packedsy,semicolon,casesy],
1791                          [ident],comma,+062);  {+063}
1792          nextif(colon1,+064);
```

```
1793          lsp:=typ(fsys+[casesy,semicolon]);
1794          if lsp<>nil then if withfile in lsp^.sflag then
1795            sflag:=sflag+[withfile];
1796          while hip<>nil do
1797            begin hip^.idtype:=lsp;
1798              hip^.foffset:=address(cc,sizeof(lsp),spack in sflag);
1799              hip:=hip^.next
1800            end;
1801        until lastsemicolon(fsys+[casesy],[ident],+065);  {+066 +067}
1802      if sy=casesy then fldlist:=varpart(fsys) else fldlist:=nil;
1803    end;


1806    begin  {typ}
1807      sflag:=[]; lsp:=nil;
1808      if sy=packedsy then begin sflag:=[spack]; insym end;
1809      if find1([ident..filesy],fsys,+068) then
1810        if sy in [ident..arrow] then
1811          begin if spack in sflag then error(+069);
1812            if sy=arrow then
1813              begin lsp:=newsp(pointer,ptrsize); insym;
1814                if not intypedec then lsp^.eltype:=stypid(+070) else
1815                  if sy<>ident then error(+071) else
1816                    begin fwptr:=newip(types,id,lsp,fwptr); insym end
1817              end
1818            else lsp:=simpletyp(fsys);
1819          end
1820        else
1821          case sy of
1822    {<<<<<<<<<<<<}
1823    arraysy:
1824      lsp:=arraytyp(fsys,arrays,sflag,typ);
1825    recordsy:
1826      begin insym;
1827        new(lsp,rec); lsp^.occur:=rec; lsp^.mlink:=top;
1828        lsp^.fname:=nil; top:=lsp;
1829        oc:=0; lsp1:=fldlist(fsys+[endsy]);  {fldlist updates oc}
1830        lsp:=newsp(records,oc); lsp^.tagsp:=lsp1;
1831        lsp^.fstfld:=top^.fname; lsp^.sflag:=sflag;
1832        top:=top^.mlink; nextif(endsy,+072)
1833      end;
1834    setsy:
1835      begin insym; nextif(ofsy,+073); lsp1:=simpletyp(fsys);
1836        if bounds(lsp1,min,max) then lsp1:=desub(lsp1) else
1837          if lsp1=intptr then
1838            begin error(-(+074)); max:=iopt-1 end
1839          else
1840            begin error(+075); lsp1:=nil end;
1841        if lsp1=intptr then sz:=iopt-1 else
1842          begin if bounds(lsp1,min,max) then {nothing}; sz:=max end;
1843        if (min<0) or (max>sz) or (sz div bytebits >= maxsetsize) then
1844          begin error(+076); lsp1:=nil; sz:=0 end;
1845        lsp:=newsp(power,sz div bytebits +1); lsp^.elset:=lsp1;
1846      end;
1847    filesy:
1848      begin insym; nextif(ofsy,+077); lsp1:=typ(fsys);
```

```
1849        if lsp1<>nil then if withfile in lsp1^.sflag then error(-(+078));
1850        sz:=sizeof(lsp1); if sz<buffsize then sz:=buffsize;
1851        lsp:=newsp(files,sz+fhsize); lsp^.filtype:=lsp1;
1852      end;
1853    {>>>>>>>>>>>>}
1854          end;  {case}
1855      typ:=lsp;
1856    end;

1858    function vpartyp(fsys:sos):sp;
1859    begin
1860      if find2([arraysy],fsys+[ident],+079) then
1861        vpartyp:=arraytyp(fsys,carray,[],vpartyp)
1862      else
1863        vpartyp:=stypid(+080)
1864    end;

1866    {=============================================================}

1868    procedure block(fsys:sos; fip:ip); forward;
1869            {pfdeclaration calls block. With a more obscure lexical
1870              structure this forward declaration can be avoided}

1872    procedure labeldeclaration(fsys:sos);
1873    var llp:lp;
1874    begin with b do begin
1875      repeat
1876        if sy<>intcst then error(+081) else
1877          begin
1878            if searchlab(lchain,val)<>nil then errint(+082,val) else
1879              begin new(llp); llp^.labval:=val;
1880                if val>9999 then teststandard;
1881                ilbno:=ilbno+1; llp^.labname:=ilbno; llp^.labdlb:=0;
1882                llp^.seen:=false; llp^.nextlp:=lchain; lchain:=llp;
1883              end;
1884            insym
1885          end
1886      until endofloop(fsys+[semicolon],[intcst],comma,+083);  {+084}
1887      nextif(semicolon,+085)
1888    end end;

1890    procedure constdefinition(fsys:sos);
1891    var lip:ip;
1892    begin
1893      repeat lip:=newident(konst,nil,nil,+086);
1894        if lip<>nil then
1895          begin nextif(eqsy,+087);
1896            constant(fsys+[semicolon,ident],lip^.idtype,lip^.value);
1897            nextif(semicolon,+088); enterid(lip);
1898          end;
1899      until not find2([ident],fsys,+089);
1900    end;

1902    procedure typedefinition(fsys:sos);
1903    var lip:ip;
1904    begin fwptr:=nil; intypedec:=true;
```



```
1905      repeat lip:=newident(types,nil,nil,+090);
1906        if lip<>nil then
1907          begin nextif(eqsy,+091);
1908            lip^.idtype:=styp(fsys+[semicolon,ident]);
1909            nextif(semicolon,+092); enterid(lip);
1910          end;
1911      until not find2([ident],fsys,+093);
1912      while fwptr<>nil do
1913        begin assert sy<>ident;
1914          id:=fwptr^.name; lip:=searchid([types]);
1915          fwptr^.idtype^.eltype:=lip^.idtype; fwptr:=fwptr^.next
1916        end;
1917      intypedec:=false;
1918    end;

1920    procedure vardeclaration(fsys:sos);
1921    var lip,hip,vip:ip; lsp:sp;
1922    begin with b do begin
1923      repeat hip:=nil; lip:=nil;
1924        repeat vip:=newident(vars,nil,nil,+094);
1925          if vip<>nil then
1926            begin enterid(vip); vip^.iflag:=[];
1927              if lip=nil then hip:=vip else lip^.next:=vip; lip:=vip;
1928            end;
1929        until endofloop(fsys+[colon1,ident..packedsy],[ident],comma,+095);
1930                                                                {+096}
1931        nextif(colon1,+097);
1932        lsp:=typ(fsys+[semicolon,ident]);
1933        while hip<>nil do
1934          begin hip^.idtype:=lsp;
1935            hip^.vpos.ad:=address(lc,sizeof(lsp),false); hip:=hip^.next
1936          end;
1937        nextif(semicolon,+098);
1938      until not find2([ident],fsys,+099);
1939    end end;

1941    procedure pfhead(fsys:sos;
1942                     var fip:ip;
1943                     var again:boolean;
1944                     parm:boolean); forward;

1946    function parlist(fsys:sos; var klc:integer):ip;
1947    var lastip,hip,lip,pip:ip; lsp,tsp:sp; iflag:iflagset; again:boolean;
1948        sz:integer;
1949    begin parlist:=nil; lastip:=nil;
1950      repeat {comes for each formal-parameter-section}
1951        if find1([ident,varsy,procsy,funcsy],fsys+[semicolon],+0100) then
1952          begin
1953            if (sy=procsy) or (sy=funcsy) then
1954              begin
1955                pfhead(fsys+[semicolon,ident,varsy,procsy,funcsy],
1956                       hip,again,true);
1957                hip^.pfpos.ad:=address(klc,paramsize+ptrsize,false);
1958                hip^.pfkind:=formal; lip:=hip;
1959                top:=top^.mlink; level:=level-1
1960              end
```

```
1961            else
1962              begin hip:=nil; lip:=nil; iflag:=[assigned,noreg];
1963                if sy=varsy then
1964                  begin iflag:=[refer,assigned,used,noreg]; insym end;
1965                repeat pip:=newident(vars,nil,nil,+0101);
1966                  if pip<>nil then
1967                    begin enterid(pip); pip^.iflag:=iflag;
1968                      if lip=nil then hip:=pip else lip^.next:=pip;
1969                      lip:=pip;
1970                    end;
1971                  iflag:=iflag+[samesect];
1972                until endofloop(fsys+[semicolon,colon1],
1973                                [ident],comma,+0102);  {+0103}
1974                nextif(colon1,+0104);
1975                if refer in iflag then
1976                  begin lsp:=vpartyp(fsys+[semicolon]);
1977                    sz:=ptrsize; tsp:=lsp;
1978                    while formof(tsp,[carray]) do
1979                      begin tsp^.arpos.ad:=address(klc,ptrsize,false);
1980                        tsp:=tsp^.eltype
1981                      end;
1982                  end
1983                else
1984                  begin lsp:=stypid(+0105); sz:=sizeof(lsp) end;
1985                pip:=hip;
1986                while pip<>nil do
1987                  begin pip^.vpos.ad:=address(klc,sz,false);
1988                    pip^.idtype:=lsp; pip:=pip^.next
1989                  end;
1990              end;
1991            if lastip=nil then parlist:=hip else lastip^.next:=hip;
1992            lastip:=lip;
1993          end;
1994      until endofloop(fsys,[ident,varsy,procsy,funcsy],
1995                      semicolon,+0106);  {+0107}
1996    end;

1998    procedure pfhead;  {forward declared}
1999    var lip:ip; lsp:sp; lmp:mp; kl:idclass;
2000    begin lip:=nil; again:=false;
2001      if sy=procsy then kl:=proc else
2002        begin kl:=func; fsys:=fsys+[colon1,ident] end;
2003      insym;
2004      if sy<>ident then begin error(+0108); id:=spaces end;
2005      if not parm then lip:=searchsection(top^.fname);
2006      if lip<>nil then
2007        if (lip^.klass<>kl) or (lip^.pfkind<>forward) then
2008          errid(+0109,id)
2009        else
2010          begin b.forwcount:=b.forwcount-1; again:=true end;
2011      if again then insym else
2012        begin lip:=newip(kl,id,nil,nil);
2013          if sy=ident then begin enterid(lip); insym end;
2014          lastpfno:=lastpfno+1; lip^.pfno:=lastpfno;
2015        end;
2016      level:=level+1;
```

```
2017    new(lnp,blok); lnp^.occur:=blck; lnp^.nlink:=top; top:=lnp;
2018    if again then lnp^.fname:=lip^.perhead else
2019      begin lnp^.fname:=nil;
2020        if find3(lparent,fsys,+0110) then
2021          begin lip^.perhead:=parlist(fsys+[rparent],lip^.headlc);
2022            nextif(rparent,+0111)
2023          end;
2024      end;
2025    if (kl=func) and not again then
2026      begin nextif(colon!,+0112); lsp:=stypid(+0113);
2027        if formof(lsp,[power..tag]) then
2028          begin error(+0114); lsp:=nil end;
2029        lip^.idtype:=lsp;
2030      end;
2031    fip:=lip;
2032  end;

2034  procedure pfdeclaration(fsys:sos);
2035  var lip:ip; again:boolean; markp:^integer; lbp:bp;
2036  begin with b do begin
2037    pfhead(fsys+[ident,semicolon,labelsy..beginsy],lip,again,false);
2038    nextif(semicolon,+0115);
2039    if find!([ident,labelsy..beginsy],fsys+[semicolon],+0116) then
2040      if sy=ident then
2041        if id='forward ' then
2042          begin insym;
2043            if lip^.pfpos.lv>1 then genpnam(ps_fwp,lip);
2044            if again then errid(+0117,lip^.name) else
2045              begin lip^.pfkind:=forwrd; forwcount:=forwcount+1 end;
2046          end else
2047        if id='extern  ' then
2048          begin lip^.pfkind:=extrn;
2049            lip^.pfpos.lv:=1; insym; teststandard
2050          end
2051        else errid(+0118,id)
2052      else
2053        begin lip^.pfkind:=actual;
2054  {$ifndef STANDARD
2055        mark(markp);
2056  {$endif
2057        if not again then if lip^.pfpos.lv>1 then genpnam(ps_fwp,lip);
2058        new(lbp); lbp^:=b; nextbp:=lbp;
2059        lc:=address(lip^.headlc,0,false);  {align headlc}
2060        ilbase:=0; forwcount:=0; lchain:=nil;
2061        if lip^.idtype<>nil then
2062          lip^.pfpos.ad:=address(lc,sizeof(lip^.idtype),false);
2063        block(fsys+[semicolon],lip);
2064        b:=nextbp^;
2065  {$ifndef STANDARD
2066        release(markp);
2067  {$endif
2068        end;
2069    if not main then enterspected:=forwcount=0;
2070    nextif(semicolon,+0119);
2071    level:=level-1; top:=top^.nlink;
2072  end end;

2074  {*************************************************************}

2076  procedure expression(fsys:sos); forward;
2077          {this forward declaration cannot be avoided}

2079  procedure selectarrayelement(fsys:sos);
2080  var isp,lsp:sp;
2081  begin
2082    repeat loadaddr; isp:=nil;
2083      if formof(a.asp,[arrays,carray]) then isp:=a.asp^.inxtype else
2084        asperr(+0120);
2085      lsp:=a.asp;
2086      expression(fsys+[comma]); force(desub(isp),+0121);
2087              {no range check}
2088      if lsp<>nil then
2089        begin a.packbit:=spack in lsp^.sflag;
2090          desoraddr(lsp^.arpos); lsp:=lsp^.eltype
2091        end;
2092      a.asp:=lsp; a.ak:=indexed;
2093    until endofloop(fsys,[notsy..lparent],comma,+0122);  {+0123}
2094  end;

2096  procedure selector(fsys: sos; fip:ip; iflag:iflagset);
2097  {selector computes the address of any kind of variable.
2098    Four possibilities:
2099    1.for direct accessable variables, 'a' contains offset and level,
2100    2.for indirect accessable variables, the address is on the stack.
2101    3.for array elements, the top of stack gives the index (one word).
2102      The address of the array is beneath it.
2103    4.for variables with address in direct accessible pointer variable,
2104      the offset and level of the pointer is stored in 'a'.
2105    If a.asp=nil then an error occurred else a.asp gives
2106    the type of the variable.
2107  }
2108  var lip:ip; l1,l2:integer;
2109  begin l1:=line; init(fip^.idtype,0);
2110    case fip^.klass of
2111      vars: with a do
2112        begin pos:=fip^.vpos; if refer in fip^.iflag then ak:=pfixed end;
2113      field:
2114        begin a:=lmstmp^.va;
2115          fieldaddr(fip^.foffset); a.asp:=fip^.idtype
2116        end;
2117      func: with a do
2118        if fip^.pfkind=standard then asperr(+0124) else
2119          begin pos:=fip^.pfpos; pos.lv:=pos.lv+1;
2120            if pos.lv>level then if fip<>currproc then error(+0125);
2121            if fip^.pfkind<>actual then error(+0126);
2122            if sy=arrow then error(+0127);
2123          end
2124    end;  {case}
2125    while find2([lbrack,period,arrow],fsys,+0128) do with a do
2126      if sy=lbrack then
2127        begin insym;
2128          selectarrayelement(fsys+[rbrack,lbrack,period,arrow]);
```



```
2129          nextif(rbrack,+0129); iflag:=iflag+[noreg];
2130        end else
2131      if sy=period then
2132        begin insym; iflag:=iflag+[noreg];
2133          if sy<>ident then error(+0130) else
2134            begin
2135              if not formof(asp,[records]) then asperr(+0131) else
2136                begin lip:=searchsection(asp^.fstfld);
2137                  if lip=nil then begin errid(+0132,id); asp:=nil end else
2138                    if lip=nil then begin errid(+0132,id); asp:=nil end else
2139                      begin packbit:=spack in asp^.sflag;
2140                        fieldaddr(lip^.foffset); asp:=lip^.idtype
2141                      end
2142                end;
2143              insym
2144            end
2145        end
2146      else
2147        begin insym; iflag:=[used];
2148          if asp<>nil then
2149            if asp=stringptr then asperr(+0133) else
2150            if asp^.form=pointer then
2151              begin
2152                if ak=pfixed then ak:=pfixed else
2153                  begin load; ak:=ploaded end;
2154                asp:=asp^.eltype
2155              end else
2156            if asp^.form=files then
2157              begin l2:=line; gen1(op_mrk,0); exchange(l1,l2); loadaddr;
2158                gencup(VAL); asp:=asp^.filtype; ak:=ploaded; packbit:=true;
2159              end
2160            else asperr(+0134);
2161        end;
2162    fip^.iflag:=fip^.iflag+ifl=gl
2163  end;

2164  procedure variable(fsys:sos);
2165  var lip: ip;
2166  begin
2167    if sy=ident then
2168      begin lip:=searchid([vars,field]); insym;
2169        selector(fsys,lip,[used,assigned,noreg])
2170      end
2171    else begin error(+0135); init(a,0) end;
2172  end;

2174  {*************************************************************}

2176  function plistequal(p1,p2:ip):boolean;
2177  var ok:boolean; q1,q2:sp;
2178  begin plistequal:=eqstruct(p1^.idtype,p2^.idtype);
2179    p1:=p1^.perhead; p2:=p2^.perhead;
2180    while (p1<>nil) and (p2<>nil) do
2181      begin ok:=false;
2182        if p1^.klass=p2^.klass then
2183          if p1^.klass<>vars then ok:=plistequal(p1,p2) else
2184            begin q1:=p1^.idtype; q2:=p2^.idtype; ok:=true;
2185              while ok and formof(q1,[carray]) and formof(q2,[carray]) do
2186                begin ok:=eqstruct(q1^.inxtype,q2^.inxtype);
2187                  q1:=q1^.eltype; q2:=q2^.eltype;
2188                end;
2189              if not (eqstruct(q1,q2) and
2190                  (p1^.iflag*[refer,samesect]=p2^.iflag*[refer,samesect]))
2191                then ok:=false;
2192            end;
2193        if not ok then plistequal:=false;
2194        p1:=p1^.next; p2:=p2^.next
2195      end;
2196    if (p1<>nil) or (p2<>nil) then plistequal:=false
2197  end;

2199  procedure callnonstandard(fsys:sos; moreargs:boolean; fip:ip);
2200  var nxt,lip:ip; lpos:position; l1,l2:integer;
2201      lsp,oldasp:sp;
2202  begin with a,lpos do begin
2203    nxt:=fip^.perhead; lpos:=fip^.pfpos;
2204    if fip^.pfkind<>formal then gen1(op_mrk,level-lv) else
2205      begin lexical(op_loi,lv,ad,ptrsize); gen0(op_mrs) end;
2206    while (nxt<>nil) and moreargs do
2207      begin lsp:=nxt^.idtype;
2208        if nxt^.klass=vars then
2209          if refer in nxt^.iflag then  {call by reference}
2210            begin l1:=line; variable(fsys); loadaddr;
2211              if samesect in nxt^.iflag then lsp:=oldasp else
2212                begin oldasp:=asp; l2:=line;
2213                  while formof(lsp,[carray]) and
2214                      formof(asp,[arrays,carray]) do
2215                    if (compt(lsp^.inxtype,asp^.inxtype) > subeq) or
2216                        (lsp^.sflag<>asp^.sflag) then asperr(+0136) else
2217                      begin desoraddr(asp^.arpos);
2218                        asp:=asp^.eltype; lsp:=lsp^.eltype
2219                      end;
2220                  exchange(l1,l2);
2221                end;
2222              if not eqstruct(asp,lsp) then asperr(+0137);
2223              if packbit then asperr(+0138);
2224            end
2225          else  {call by value}
2226            begin expression(fsys); force(lsp,+0139) end
2227        else
2228          if sy<>ident then error(+0140) else
2229            begin lip:=searchid([nxt^.klass]); insym;
2230              if lip^.pfkind=standard then error(+0141) else
2231              if not plistequal(nxt,lip) then error(+0142) else
2232              if lip^.pfkind=formal then
2233                lexical(op_loi,lip^.pfpos.lv,
2234                        lip^.pfpos.ad,psumsize+ptrsize)
2235              else
2236                begin gen1(op_lex,level-lip^.pfpos.lv);
2237                  genpnam(op_loe,lip)
2238                end
2239            end;
2240        nxt:=nxt^.next; moreargs:=find3(comma,fsys,+0143);
```

```
2241        end;
2242      while moreargs do
2243        begin error(+0144); expression(fsys); load;
2244          moreargs:=find3(comma,fsys,+0145)
2245        end;
2246      if xrt<>nil then error(+0146);
2247      if fip^.pfkind<>formal then genpnam(op_cal,fip) else
2248        begin lexical(op_loi,lv,ad+ptrsize,pnumsize); gen0(op_cas) end;
2249      asp:=fip^.idtype;
2250    end end;

2252    procedure fileaddr;
2253    var la:attr;
2254    begin la:=a; a:=fa; loadaddr; a:=la end;

2256    procedure callr(l1,l2:integer);
2257    var la:attr;
2258    begin with a do begin
2259      la:=a; asp:=desub(asp); gen1(op_mrk,0); fileaddr;
2260      if asp=intptr then gensp(RDI) else
2261      if asp=charptr then gensp(RDC) else
2262      if asp=realptr then gensp(RDR) else
2263      if asp=longptr then gensp(RDL) else asperr(+0147);
2264      if asp<>la.asp then checkbnds(la.asp);
2265      a:=la; exchange(l1,l2); store;
2266    end end;

2268    procedure callw(fsys:sos; l1,l2:integer);
2269    var m:libmnem;
2270    begin with a do begin gen1(op_mrk,0);
2271      fileaddr; exchange(l1,l2); loadcheap; asp:=desub(asp);
2272      if string(asp) then
2273        begin gen1(op_loc,asp^.size); m:=WRS end
2274      else
2275        begin m:=WRI;
2276          if asp<>intptr then
2277            if asp=charptr then m:=WRC else
2278            if asp=realptr then m:=WRR else
2279            if asp=boolptr then m:=WRB else
2280            if asp=stringptr then m:=WRZ else
2281            if asp=longptr then m:=WRL else asperr(+0148);
2282        end;
2283      if find3(colon1,fsys,+0149) then
2284        begin expression(fsys+[colon1]);
2285          force(intptr,+0150); m:=succ(m)
2286        end;
2287      if find3(colon1,fsys,+0151) then
2288        begin expression(fsys); force(intptr,+0152);
2289          if m<>WRI then error(+0153) else m:=WRF;
2290        end;
2291      gensp(m);
2292    end end;

2294    procedure callrw(fsys:sos; lpar,w,ln:boolean);
2295    var l1,l2,oldlc,errno:integer; ftype,lsp:sp;
2296    begin with b do begin oldlc:=lc; ftype:=textptr;

2297      inita(textptr,argv[ord(w)].ad); a.pos.lv:=0; fa:=a;
2298      if lpar then
2299        begin l1:=lino;
2300          if w then expression(fsys+[colon1]) else variable(fsys);
2301          l2:=lino;
2302          if formof(a.asp,[files]) then
2303            begin ftype:=a.asp;
2304              if (a.ak<>fixed) and (a.ak<>pfixed) then
2305                begin loadaddr; inita(nilptr,reserve(ptrsize));
2306                  store; a.ak:=pfixed
2307                end;
2308              fa:=a;  {store doesn't change a}
2309              if (sy<>comma) and not ln then error(+0154);
2310            end
2311          else
2312            begin if iop[w]=nil then error(+0155);
2313              if w then callw(fsys,l1,l2) else callr(l1,l2)
2314            end;
2315          while find3(comma,fsys,+0156) do with a do
2316            begin l1:=lino;
2317              if w then expression(fsys+[colon1]) else variable(fsys);
2318              l2:=lino;
2319              if ftype=textptr then
2320                if w then callw(fsys,l1,l2) else callr(l1,l2)
2321              else
2322                begin errno:=+0157;
2323                  if w then force(ftype^.filtype,errno) else
2324                    begin store; l2:=lino end;
2325                  gen1(op_mrk,0); fileaddr; gensp(WDW);
2326                  ak:=ploaded; packbit:=true;
2327                  if w then store else
2328                    begin lsp:=asp; asp:=ftype^.filtype; force(lsp,errno);
2329                      exchange(l1,l2)
2330                    end;
2331                  gen1(op_mrk,0); fileaddr;
2332                  if w then gensp(PUTX) else gensp(GETX)
2333                end
2334            end;
2335        end
2336      else
2337        if not ln then error(+0158) else
2338          if iop[w]=nil then error(+0159);
2339      if ln then
2340        begin if ftype<>textptr then error(+0160);
2341          gen1(op_mrk,0); fileaddr;
2342          if w then gensp(WLN) else gensp(RLN)
2343        end;
2344      lc:=oldlc
2345    end end;

2347    procedure callfip(fsys:sos; lpar:boolean; m:libmnem);
2348    begin with a do begin
2349      if lpar then
2350        begin variable(fsys); loadaddr;
2351          if not formof(asp,[files]) then asperr(+0161) else
2352            if (m<>EFL) and (asp<>textptr) then error(+0162);
```



```
2353        end
2354      else
2355        if iop[m=PAG]=nil then error(+0163) else
2356          gen1(op_lae,argv[ord(m=PAG)].ad);
2357      gensp(m); asp:=boolptr;  {not for PAG}
2358    end end;

2360    procedure callnd(fsys:sos; m:libmnem);
2361    label 1;
2362    var lsp:sp; sz,int:integer;
2363    begin with a do begin
2364      if not formof(asp,[pointer]) then asperr(+0164) else
2365        if asp=stringptr then asperr(+0165) else
2366          asp:=asp^.eltype;
2367      while find3(comma,fsys,+0166) do
2368        begin
2369          if asp<>nil then  {asp of form record or variant}
2370            if asp^.form=records then asp:=asp^.tagsp else
2371            if asp^.form=variant then asp:=asp^.subtsp else asperr(+0167);
2372          if asp=nil then constant(fsys,lsp,int) else
2373            begin assert asp^.form=tag;
2374              int:=estinteger(fsys,asp^.tfldsp,+0168); lsp:=asp^.fstvar;
2375              while lsp<>nil do
2376                if lsp^.varval<>int then lsp:=lsp^.nxtvar else
2377                  begin asp:=lsp; goto 1 end;
2378            end;
2379    1:  end;
2380      sz:=sizeof(asp); int:=intsize+ptrsize;
2381      if sz>int then int:=(sz+int-1) div int * int;
2382      gen1(op_loc,int); gensp(m)
2383    end end;

2385    procedure callpg(m:libmnem);
2386    begin gensp(m); if not formof(a.asp,[files]) then asperr(+0169) end;

2388    procedure callrr(m:libmnem);
2389    begin
2390      if not formof(a.asp,[files]) then asperr(+0170) else
2391        if a.asp=textptr then gen1(op_loc,0) else
2392          gen1(op_loc,sizeof(a.asp^.filtype));
2393      gensp(m);
2394    end;

2396    procedure callmr(m:libmnem);
2397    begin teststandard; gensp(m);
2398      if not formof(a.asp,[pointer]) then asperr(+0171)
2399    end;

2401    procedure callpu(m:libmnem; xsp,asp,lsp:sp);
2402    begin fsp:=desub(lsp);
2403      if formof(xsp,[arrays,carray]) and formof(asp,[arrays,carray]) then
2404        if (speck in (xsp^.sflag + asp^.sflag)) and
2405            eqstruct(xsp^.eltype,asp^.eltype) and
2406            eqstruct(desub(xsp^.inxtype),lsp) and
2407            eqstruct(desub(asp^.inxtype),lsp) then
2408          begin desaraddr(xsp^.arpos); desaraddr(asp^.arpos); gensp(m) end
2409        else error(+0172)
2410      else error(+0173)
2411    end;

2413    procedure call(fsys: sos; fip: ip);
2414    var lkey: standpf; lpar:boolean; lsp,lsp2:sp;
2415    begin with a do begin fsys:=fsys+[comma];
2416      lpar:=find3(lparent,fsys,+0174); if lpar then fsys:=fsys+[rparent];
2417      if fip^.pfkind<>standard then callnonstandard(fsys,lpar,fip) else
2418        begin lkey:=fip^.key;
2419          if lkey in [pput..phalt,feof..fabs,fround..farctan] then
2420            gen1(op_mrk,0);
2421          if lkey in [pput..prelease,fabs..farctan] then
2422            begin if not lpar then error(+0175);
2423              if lkey <= prelease then
2424                begin variable(fsys); loadaddr end
2425              else
2426                begin expression(fsys); force(fip^.idtype,+0176) end;
2427            end;
2428          case lkey of
2429            pread,preadln,pwrite,pwriteln:  {0,1,2,3 resp}
2430              callrw(fsys,lpar,lkey>=pwrite,odd(ord(lkey)));
2431            pput:
2432              callpg(PUTX);
2433            pget:
2434              callpg(GETX);
2435            ppage:
2436              callfip(fsys,lpar,PAG);
2437            preset:
2438              callrr(OPN);
2439            prewrite:
2440              callrr(CRE);
2441            pnew:
2442              callnd(fsys,NEWX);
2443            pdispose:
2444              callnd(fsys,DLS);
2445            ppack:
2446              begin lsp:=asp; nextif(comma,+0177); expression(fsys); load;
2447                lsp2:=asp; nextif(comma,+0178); variable(fsys); loadaddr;
2448                callpu(PAC,asp,lsp,lsp2)
2449              end;
2450            punpack:
2451              begin lsp:=asp; nextif(comma,+0179); variable(fsys); loadaddr;
2452                lsp2:=asp; nextif(comma,+0180); expression(fsys); load;
2453                callpu(UNP,lsp,lsp2,asp)
2454              end;
2455            pmark:
2456              callmr(SAV);
2457            prelease:
2458              callmr(RST);
2459            phalt:
2460              begin teststandard;
2461                if not lpar then gen1(op_loc,0) else
2462                  begin expression(fsys); force(intptr,+0181) end;
2463                gensp(HLT);
2464              end;
```

```
2465            feof:
2466              call1p(fsys,1par,EFL);
2467            feoln:
2468              call1p(fsys,1par,ELN);
2469            fabs:
2470              begin asp:=desub(asp);
2471                if asp=intptr then gensp(ABI) else
2472                if asp=realptr then gensp(ABR) else
2473                if asp=longptr then gensp(ABL) else asperr(+0182);
2474              end;
2475            fsqr:
2476              begin asp:=desub(asp);
2477                if asp=intptr then
2478                  begin gen1(op_dup,intsize); gen0(op_mul) end else
2479                if asp=realptr then
2480                  begin gen1(op_dup,realsize);
2481                    gen0(op_fmu); fltused:=true
2482                  end
2483                else if asp=longptr then
2484                  begin gen1(op_dup,longsize); gen0(op_dmu) end
2485                else asperr(+0183);
2486              end;
2487            ford:
2488              begin if not nicescalar(desub(asp)) then asperr(+0184);
2489                asp:=intptr
2490              end;
2491            fchr:
2492              checkbnds(charptr);
2493            fpred,fsucc:
2494              begin asp:=desub(asp); gen1(op_loc,1);
2495                if l=key=fpred then gen0(op_sub) else gen0(op_add);
2496                if nicescalar(asp) then genrck(asp) else asperr(+0185)
2497              end;
2498            fodd:
2499              begin gen1(op_loc,1); gen1(op_and,intsize); asp:=boolptr end;
2500            ftrunc:
2501              begin if asp<>realptr then asperr(+0186); opconvert(ri) end;
2502            fround:
2503              begin if asp<>realptr then asperr(+0187);
2504                gensp(RND); asp:=intptr
2505              end;
2506            fsin:
2507              gensp(SIN);
2508            fcos:
2509              gensp(COS);
2510            fexp:
2511              gensp(EXPX);
2512            fsqrt:
2513              gensp(SQR);
2514            fln:
2515              gensp(LOG);
2516            farctan:
2517              gensp(ATN);
2518          end;
2519        end;
2520      if lpar then nextif(rparent,+0188);
```

```
2521    end end;

2523    {*************************************************************}

2525    procedure convert(fsp:sp; l1:integer);
2526    {Convert tries to make the operands of some operator of the same type.
2527      The operand types are given by fsp and a.asp. The resulting type
2528      is put in a.asp.
2529      l1 gives the lino of the first instruction of the right operand.
2530    }
2531    var l2:integer;
2532        ta:twostruct;
2533    begin with a do begin asp:=desub(asp);
2534      ta:=compat(fsp,asp);
2535      case ta of
2536        eq,subeq:
2537          ;
2538        ri,li,ri:
2539          opconvert(compat(asp,fsp));  { ri->ir etc.}
2540        lr,ll,ir:
2541          begin l2:=lino; opconvert(ta); exchange(l1,l2) end;
2542        se:
2543          expandemptyset(fsp);
2544        es:
2545          begin l2:=lino; expandemptyset(asp); exchange(l1,l2) end;
2546        noteq:
2547          asperr(+0189);
2548      end;
2549      if asp=realptr then fltused:=true
2550    end end;

2552    procedure buildset(fsys:sos);
2553    {This is a bad construct in pascal. Two objections:
2554      - expr..expr very difficult to implement on most machines
2555      - this construct makes it hard to implement sets of different size
2556    }
2557    const   ncsw    = 16;   {tunable}
2558    type    wordset = set of 0..wbm1;
2559    var     i,j,val1,val2,ncst,l1,l2,sz:integer;
2560            cst1,cst2,cst12,varpart:boolean;
2561            cstpart:array[1..ncsw] of wordset;
2562            lsp:sp;

2564    procedure genwordset(s:wordset);
2565            {level 2: <<  buildset}
2566    var b,i,w:integer;
2567    begin
2568      if s=[] then w:=0 else
2569      if s=[wbm1] then w:=-t15m1-1 else
2570        begin w:=-1; b:=t14;
2571          for i:=wbm1-1 downto 0 do
2572            begin if i in s then w:=w+b; b:=b div 2 end;
2573          if wbm1 in s then w:=w-t15m1 else w:=w+1
2574        end;
2575      gen1(op_loc,w)
2576    end;
```



```
2578    procedure setexpr(fsys:sos; var c:boolean; var v:integer);
2579            {level 2: <<  buildset}
2580    {update lsp and sz variables of buildset and set c and v parameters}
2581    var min,max:integer; errno:integer;
2582    begin with a do begin c:=false; v:=0;
2583      expression(fsys); asp:=desub(asp);
2584      if asp<>nil then
2585        begin
2586          if lsp=nil then
2587            begin errno:=0;
2588              if not bounds(asp,min,max) then
2589                if asp=intptr then max:=iopt-1 else errno:=+0190;
2590              if max>(maxsetsize-1)*bytebits + (bytebits-1) then
2591                errno:=+0191;
2592              if errno<>0 then begin asperr(errno); max:=0 end;
2593              sz:=even(max div bytebits + 1); lsp:=asp;
2594            end
2595          else {asp<>nil and lsp<>nil}
2596            if asp<>lsp then asperr(+0192);
2597          if ak=cst then
2598            if pos.ad<ncsw*wordbits then
2599              begin c:=true; v:=pos.ad end;
2600        end;
2601      if not c then load
2602    end end;

2604    begin with a do begin  {buildset}
2605      varpart:=false; ncst:=0; sz:=maxsetsize; lsp:=nil;
2606      for i:=1 to ncsw do cstpart[i]:=[];
2607      if find2([notsy..lparent],fsys,+0193) then
2608        repeat l1:=lino;
2609          setexpr(fsys+[colon2,comma],cst1,val1); cst12:=cst1;
2610          if find2([colon2,fsys+[comma,notsy..lparent],+0194) then
2611            begin setexpr(fsys+[comma,notsy..lparent],cst2,val2);
2612              cst12:=cst1=2 and cst2;
2613              if cst2 and not cst1 then load;
2614              if cst1 and not cst2 then
2615                begin l2:=lino; gen1(op_loc,val1); exchange(l1,l2) end;
2616              if not cst12 then
2617                begin l2:=lino; gen1(op_ort,0); exchange(l1,l2);
2618                  gen1(op_loc,sz); gensp(RTS)
2619                end;
2620            end
2621          else
2622            if cst12 then val2:=val1 else gen1(op_set,sz);
2623          if cst12 then
2624            if (val1<0) or (val2>ncsw*wordbits) then error(+0195) else
2625              for i:=val1 to val2 do
2626                begin j:=i div wordbits + 1; ncst:=ncst+1;
2627                  cstpart[j]:=cstpart[j] + [i mod wordbits]
2628                end
2629          else
2630            if varpart then gen1(op_ior,sz) else varpart:=true;
2631        until endofloop(fsys,[notsy..lparent],comma,+0196);  {+0197}
2632      ak:=loaded;
```

```
2633      if (ncst=0) and not varpart then
2634        begin asp:=emptyset; gen1(op_loc,0) end
2635      else
2636        begin asp:=newsp(power,sz); asp^.elset:=lsp;
2637          if ncst>0 then
2638            for i:=1 to sz div wordsize do genwordset(cstpart[i]);
2639          if varpart and (ncst>0) then gen1(op_ior,sz);
2640        end
2641    end end;

2643    procedure fsetor(fsys: sos);
2644    var lip:ip; l1,i:integer; lsp:sp;
2645    begin with a do begin
2646      asp:=nil; packbit:=false; ak:=loaded;
2647      if find1([notsy..nilcst,lparent],fsys,+0198) then
2648        case sy of
2649          ident:
2650            begin lip:=searchid([konst,vars,field,func,carrbnd]); insym;
2651              case lip^.klass of
2652                func: {call moves result to top stack}
2653                  begin call(fsys,lip); ak:=loaded; packbit:=false end;
2654                konst:
2655                  begin asp:=lip^.idtype;
2656                    if nicescalar(asp) then  {including asp=nil}
2657                      begin ak:=cst; pos.ad:=lip^.value end
2658                    else
2659                      begin ak:=ploaded;
2660                        l1:=lino; gen0(op_lae,abs(lip^.value));
2661                        if asp^.form=scalar then
2662                          begin load; if lip^.value<0 then negate(l1) end
2663                        else
2664                          if asp=stringptr then ak:=loaded
2665                      end
2666                  end;
2667                field,vars:
2668                  selector(fsys,lip,[used]);
2669                carrbnd:
2670                  begin lsp:=lip^.idtype; assert formof(lsp,[carray]);
2671                    dmaraddr(lsp^.arpos); lsp:=lsp^.inxtype;
2672                    asp:=desub(lsp);
2673                    if lip^.next=nil then ak:=ploaded {low bound} else
2674                      begin gen1(op_loi,2*intsize); gen0(op_add) end;
2675                    load; checkbnds(lsp);
2676                  end;
2677              end  {case}
2678          end;
2679        intcst:
2680          begin asp:=intptr; ak:=cst; pos.ad:=val; insym end;
2681        realcst:
2682          begin asp:=realptr; ak:=ploaded; gend(op_lae,val); insym end;
2683        longcst:
2684          begin asp:=longptr; ak:=ploaded; gend(op_lae,val); insym end;
2685        charcst:
2686          begin asp:=charptr; ak:=cst; pos.ad:=val; insym end;
2687        stringcst;
2688          begin asp:=stringstruct; gend(op_lae,val); insym;
```

```
2689            if asp<>stringptr then ak:=ploaded;
2690          end;
2691        nilsym:
2692          begin insym; asp:=nilptr;
2693            for i:=1 to ptrsize div wordsize do gen1(op_loc,0);
2694          end;
2695        lparent:
2696          begin insym;
2697            expression(fsys+[rparent]); nextif(rparent,+0199)
2698          end;
2699        notsy:
2700          begin insym; factor(fsys); load; gen0(op_teq);
2701            if asp<>boolptr then asperr(+0200)
2702          end;
2703        lbrack:
2704          begin insym; buildset(fsys+[rbrack]); nextif(rbrack,+0201) end;
2705      end
2706    end end;

2708  procedure term(fsys:sos);
2709  var lsy:symbol; lsp:sp; l0,l1,l2:integer; first:boolean;
2710  begin with a,b do begin first:=true; l1:=lino; l0:=l1;
2711    factor(fsys+[starsy..andsy]);
2712    while find2([starsy..andsy],fsys,+0202) do
2713      begin if first then begin load; first:=false end;
2714        lsy:=sy; insym; l1:=lino; lsp:=asp;
2715        factor(fsys+[starsy..andsy]); load; convert(lsp,l1);
2716        if asp<>nil then
2717          case lsy of
2718            starsy:
2719              if asp=intptr then gen0(op_mul) else
2720              if asp=realptr then gen0(op_fmu) else
2721              if asp=longptr then gen0(op_dmu) else
2722              if asp^.form=power then setop(op_and) else asperr(+0203);
2723            slashsy:
2724              if asp=realptr then gen0(op_fdv) else
2725              if (asp=intptr) or (asp=longptr) then
2726                begin lsp:=asp;
2727                  convert(realptr,l1);  {make real of right operand}
2728                  convert(lsp,l1);  {make real of left operand}
2729                  gen0(op_fdv)
2730                end
2731              else asperr(+0204);
2732            divsy:
2733              if asp=intptr then gen0(op_div) else
2734              if asp=longptr then gen0(op_ddv) else asperr(+0205);
2735            modsy:
2736              begin l2:=lino; gen1(op_mrk,0); exchange(l0,l2);
2737                if asp=intptr then gensp(MDI) else
2738                if asp=longptr then gensp(MDL) else asperr(+0206);
2739              end;
2740            andsy:
2741              if asp=boolptr then setop(op_and) else asperr(+0207);
2742          end {case}
2743      end {while}
2744  end end;
```

```
2746  procedure simpleexpression(fsys:sos);
2747  var lsy:symbol; lsp:sp; l1:integer; signed,min,first:boolean;
2748  begin with a do begin l1:=lino; first:=true;
2749    signed:=(sy=plussy) or (sy=minsy);
2750    if signed then begin min:=sy=minsy; insym end else min:=false;
2751    term(fsys + [minsy,plussy,orsy]); lsp:=desub(asp);
2752    if signed then
2753      if (lsp<>intptr) and (lsp<>realptr) and (lsp<>longptr) then
2754        asperr(+0208)
2755      else if min then
2756        begin load; first:=false; asp:=lsp; negate(l1) end;
2757    while find2([plussy,minsy,orsy],fsys,+0209) do
2758      begin if first then begin load; first:=false end;
2759        lsy:=sy; insym; l1:=lino; lsp:=asp;
2760        term(fsys+[minsy,plussy,orsy]); load; convert(lsp,l1);
2761        if asp<>nil then
2762          case lsy of
2763            plussy:
2764              if asp=intptr then gen0(op_add) else
2765              if asp=realptr then gen0(op_fad) else
2766              if asp=longptr then gen0(op_dad) else
2767              if asp^.form=power then setop(op_ior) else asperr(+0210);
2768            minsy:
2769              if asp=intptr then gen0(op_sub) else
2770              if asp=realptr then gen0(op_fsb) else
2771              if asp=longptr then gen0(op_dsb) else
2772              if asp^.form=power then
2773                begin setop(op_com); setop(op_and) end
2774              else asperr(+0211);
2775            orsy:
2776              if asp=boolptr then setop(op_ior) else asperr(+0212);
2777          end {case}
2778      end {while}
2779  end end;

2781  procedure expression; { fsys:sos }
2782  var lsy:symbol; lsp:sp; l1,l2,l3,sz:integer;
2783  begin with a do begin l1:=lino;
2784    simpleexpression(fsys+[eqsy..insy]);
2785    if find2([eqsy..insy],fsys,+0213) then
2786      begin lsy:=sy; insym; lsp:=asp; loadcheap; l2:=lino;
2787        simpleexpression(fsys); loadcheap;
2788        if lsy=insy then
2789          begin
2790            if not formof(asp,[power]) then asperr(+0214) else
2791              if asp=emptyset then setop(op_and) else
2792                    {this effectively replaces the word on top  the
2793                     stack by the result of the 'in' operator.  e }
2794              if not (compat(lsp,asp^.elset) <= subeq) then
2795                asperr(+0215)
2796              else
2797                begin exchange(l1,l2); setop(op_inn) end
2798          end
2799        else
2800          begin convert(lsp,l2);
```



```
2801            if asp<>nil then
2802              case asp^.form of
2803                scalar:
2804                  if asp=realptr then gen0(op_cmf) else
2805                  if asp=longptr then gen0(op_cmd) else gen0(op_cmi);
2806                pointer:
2807                  if (lsy=eqsy) or (lsy=nesy) then gen0(op_cmp) else
2808                    asperr(+0216);
2809                power:
2810                  case lsy of
2811                    eqsy,nesy: setop(op_cmu);
2812                    ltsy,gtsy: asperr(+0217);
2813                    lesy:  {'a<=b' equivalent to 'a-b=[]'}
2814                      begin setop(op_com); setop(op_and);
2815                        gen1(op_loc,0); expandemptyset(asp);
2816                        setop(op_cmu); lsy:=eqsy
2817                      end;
2818                    gesy:  {'a>=b' equivalent to 'a=b+a'}
2819                      begin sz:=even(sizeof(asp)); gen1(op_dup,2*sz);
2820                        gen1(op_beg,-sz); setop(op_ior);
2821                        setop(op_cmu); lsy:=eqsy
2822                      end
2823                  end;  {case}
2824                arrays:
2825                  if string(asp) then
2826                    begin l3:=lino; gen1(op_mrk,0); exchange(l1,l3);
2827                      gen1(op_loc,asp^.size); gensp(BCP)
2828                    end
2829                  else asperr(+0218);
2830                records: asperr(+0219);
2831                files: asperr(+0220)
2832              end;  { case }
2833            case lsy of
2834              ltsy: gen0(op_tlt);
2835              lesy: gen0(op_tle);
2836              gtsy: gen0(op_tgt);
2837              gesy: gen0(op_tge);
2838              nesy: gen0(op_tne);
2839              eqsy: gen0(op_teq)
2840            end
2841          end;
2842        asp:=boolptr; ak:=loaded
2843      end;
2844  end end;

2846  {*****************************************************************}

2848  procedure statement(fsys:sos); forward;
2849                    {this forward declaration can be avoided}

2851  procedure assignment(fsys:sos; fip:ip);
2852  var la:attr; l1,l2:integer;
2853  begin
2854    l1:=lino; selector(fsys+[becomes],fip,[assigned]); l2:=lino;
2855    la:=a; nextif(becomes,+0221);
2856    expression(fsys); loadcheap; checkasp(la.asp,+0222);
```

```
2857    exchange(l1,l2); a:=la;
2858    if not formof(la.asp,[arrays..records]) then store else
2859      begin loadaddr;
2860        if la.asp^.form<>carray then
2861          gen1(op_blm,even(sizeof(la.asp)))
2862        else
2863          begin gen1(op_mrk,0); descraddr(la.asp^.arpos); gensp(ASZ);
2864            gen0(op_bls)
2865          end;
2866      end;
2867  end;

2869  procedure gotostatement;
2870  {jumps into structured statements can give strange results. }
2871  label 1;
2872  var llp:lp; lbp:bp; diff:integer;
2873  begin
2874    if sy<>intcst then error(+0223) else
2875      begin llp:=searchlab(b,lchain,val);
2876        if llp<>nil then
2877          if llp^.seen then gen1(op_brb,llp^.labname)
2878          else gen1(op_brf,llp^.labname)
2879        else
2880          begin lbp:=b.nextbp; diff:=1;
2881            while lbp<>nil do
2882              begin llp:=searchlab(lbp^.lchain,val);
2883                if llp<>nil then goto 1;
2884                lbp:=lbp^.nextbp; diff:=diff+1;
2885              end;
2886            if llp=nil then errint(+0224,val) else
2887  1:          begin
2888                if llp^.labdlbx=0 then
2889                  begin dlbno:=dlbno+1; llp^.labdlb:=dlbno;
2890                    gend(pe_fwa,dlbno);  {forward data reference}
2891                  end;
2892                gen1(op_mrk,diff); gend(op_lae,llp^.labdlb); gensp(GTO);
2893              end;
2894          end;
2895        insym;
2896      end
2897  end;

2899  procedure compoundstatement(fsys:sos; err:integer);
2900  begin
2901    repeat statement(fsys+[semicolon])
2902    until endofloop(fsys,[beginsy..casesy],semicolon,err)
2903  end;

2905  procedure ifstatement(fsys:sos);
2906  var lb1,lb2:integer;
2907  begin with b do begin
2908    expression(fsys+[thensy,elsesy]);
2909    force(boolptr,+0225); ilbno:=ilbno+1; lb1:=ilbno; gen1(op_zeq,lb1);
2910    nextif(thensy,+0226); statement(fsys+[elsesy]);
2911    if find3(elsesy,fsys,+0227) then
2912      begin ilbno:=ilbno+1; lb2:=ilbno; gen1(op_brf,lb2);
```

```
2913          genilb(lb1); statement(fsys); genilb(lb2)
2914        end
2915      else genilb(lb1);
2916    end end;

2918    procedure casestatement(fsys:sos);
2919    label 1;
2920    type cip=^caseinfo,
2921         caseinfo=record
2922           next: cip;
2923           csstart: integer;
2924           cslab: integer
2925         end;
2926    var lsp:sp; head,p,q,r:cip;
2927        l0,l1,l2,i,n,m,min,max:integer;
2928    begin with b do begin
2929      expression(fsys+[ofsy,semicolon,ident..plussy]); lsp:=s.asp; load;
2930      if not nicescalar(desub(lsp)) then begin error(+0228); lsp:=nil end;
2931      ilbno:=ilbno+1; l0:=ilbno; gen1(op_brf,l0);  {jump to CSA/B}
2932      ilbno:=ilbno+1; l1:=ilbno;
2933      nextif(ofsy,+0229); head:=nil; max:=minint; min:=maxint; n:=0;
2934      repeat ilbno:=ilbno+1; l2:=ilbno;   {label of current case}
2935        repeat i:=cstinteger(fsys+[comma,colon1,semicolon],lsp,+0230);
2936          if i>max then max:=i; if i<min then min:=i; n:=n+1;
2937          q:=head; r:=nil; new(p);
2938          while q<>nil do
2939            begin  {chain all cases in ascending order}
2940              if q^.cslab>=i then
2941                begin if q^.cslab=i then error(+0231); goto 1 end;
2942              r:=q; q:=q^.next
2943            end;
2944    1:    p^.next:=q; p^.cslab:=i; p^.csstart:=l2;
2945          if r=nil then head:=p else r^.next:=p;
2946        until endofloop(fsys+[colon1,semicolon],
2947                        [ident..plussy],comma,+0232);  {+0233}
2948      nextif(colon1,+0234); genilb(l2); statement(fsys+[semicolon]);
2949      gen1(op_brf,l1);
2950    until lastsemicolon(fsys,[ident..plussy],+0235);  {+0236 +0237}
2951    assert n<>0;
2952    dlbno:=dlbno+1; gendlb(dlbno); genpsam(ps_rom,currproc); gencst(-1);
2953    if (max div 3) - (min div 3) < n then
2954      begin gencst(min); gencst(max-min);
2955        m:=op_csa;
2956        while head<>nil do
2957          begin
2958            while head^.cslab>min do
2959              begin gencst(-1); min:=min+1 end;
2960            gencslb(head^.csstart); min:=min+1; head:=head^.next
2961          end;
2962      end
2963    else
2964      begin gencst(n); m:=op_csb;
2965        while head<>nil do
2966          begin gencst(head^.cslab);
2967            gencslb(head^.csstart); head:=head^.next
2968          end;
2969      end;
2970    genend; genilb(l0); gend(op_lae,dlbno); gen0(m); genilb(l1)
2971  end end;

2973  procedure repeatstatement(fsys:sos);
2974  var lb1: integer;
2975  begin with b do begin
2976    ilbno:=ilbno+1; lb1:=ilbno; genilb(lb1);
2977    compoundstatement(fsys+[untilsy],+0238);  {+0239}
2978    nextif(untilsy,+0240); genlin;
2979    expression(fsys); force(boolptr,+0241);
2980    ilbno:=ilbno+1; gen0(op_teq); gen1(op_zeq,ilbno);
2981    gen1(op_brb,lb1); genilb(ilbno)
2982  end end;

2984  procedure whilestatement(fsys:sos);
2985  var lb1,lb2: integer;
2986  begin with b do begin
2987    ilbno:=ilbno+2; lb1:=ilbno-1; genilb(lb1); lb2:=ilbno;
2988    genlin; expression(fsys+[dosy]);
2989    force(boolptr,+0242); gen1(op_zeq,lb2);
2990    nextif(dosy,+0243); statement(fsys);
2991    gen1(op_brb,lb1); genilb(lb2)
2992  end end;

2994  procedure forstatement(fsys:sos);
2995  {the upper bound is evaluated once and stored in a temporary local}
2996  var lip:ip; dsp,lsp:sp; tosym,cst1,cst2,local:boolean;
2997      val1,val2,endlab,looplab,oldlc,llc,lad:integer;
2998  begin with s,b do begin
2999    lsp:=nil; lad:=0; tosym:=true; local:=level<>1; oldlc:=lc;
3000    ilbno:=ilbno+1; looplab:=ilbno; ilbno:=ilbno+1; endlab:=ilbno;
3001    if sy<>ident then error(+0244) else
3002      begin lip:=searchid([vars]); insym;
3003        lsp:=lip^.idtype; lad:=lip^.vpos.ad;
3004        if local and
3005            ((lad<currproc^.headlc) or (lip^.vpos.lv<>level)) then
3006          error(+0245)
3007        else lip^.iflag:=lip^.iflag+[used,assigned];
3008      end;
3009    if not nicescalar(desub(lsp)) then begin error(+0246); lsp:=nil end;
3010    nextif(becomes,+0247); dsp:=desub(lsp); assert sizeof(dsp)=wordsize;
3011    expression(fsys+[tosy,downtosy,notsy..lparent,dosy]);
3012    cst1:=s.k=cst; if cst1 then val1:=s.pos.ad; force(dsp,+0248);
3013    if not cst1 then gen1(op_dup,intsize);
3014    if find1([tosy,downtosy],fsys+[notsy..lparent,dosy],+0249) then
3015      begin tosym:=sy=tosy; insym end;
3016    expression(fsys+[dosy]);
3017    cst2:=s.k=cst; if cst2 then val2:=s.pos.ad; force(dsp,+0250);
3018    if not cst2 then
3019      begin llc:=reserve(intsize);
3020        gen1(op_dup,intsize); gen1(op_stl,llc);
3021      end;
3022    if cst1 then
3023      begin
3024        if tosym then gen1(op_bgt,endlab) else gen1(op_blt,endlab);
```



```
3025          gen1(op_loc,val1)
3026        end
3027      else
3028        begin ilbno:=ilbno+1;
3029          if tosym then gen1(op_ble,ilbno) else gen1(op_bge,ilbno);
3030          gen1(op_beq,-intsize); gen1(op_brf,endlab); genilb(ilbno)
3031        end;
3032      assert eqstruct(s.asp,dsp);
3033      checkbnds(lsp); pop(local,lad,intsize); genilb(looplab);
3034      nextif(dosy,+0251); statement(fsys);
3035      push(local,lad,intsize);
3036      if cst2 then gen1(op_loc,val2) else gen1(op_lol,llc);
3037      gen1(op_beq,endlab); push(local,lad,intsize); gen1(op_loc,1);
3038      if tosym then gen0(op_add) else gen0(op_sub);
3039      s.asp:=dsp; checkbnds(lsp); pop(local,lad,intsize);
3040      gen1(op_brb,looplab); genilb(endlab);
3041      lc:=oldlc
3042    end end;

3044    procedure withstatement(fsys:sos);
3045    var lnp,oldtop:np; oldlc:integer; pbit:boolean;
3046    begin with b do begin
3047      oldlc:=lc; oldtop:=top;
3048      repeat variable(fsys+[comma,dosy]);
3049        if not formof(s.asp,[records]) then asperr(+0252) else
3050          begin pbit:=speak in s.asp^.aflag;
3051            new(lnp,wrec); lnp^.occur:=wrec; lnp^.fname:=s.asp^.fstfld;
3052            if s.ak<>fixed then
3053              begin loadaddr; incita(nilptr,reserve(ptrsize)); store;
3054                s.ak:=pfixed;
3055              end;
3056            s.packbit:=pbit; lnp^.wa:=s; lnp^.nlink:=top; top:=lnp;
3057          end;
3058      until endofloop(fsys+[dosy],[ident],comma,+0253);  {+0254}
3059      nextif(dosy,+0255); statement(fsys);
3060      top:=oldtop; lc:=oldlc;
3061    end end;

3063    procedure assertion(fsys:sos);
3064    begin teststandard;
3065      if opt['a']=off then
3066        while not (sy in fsys) do insym
3067      else
3068        begin gen1(op_mrk,0); expression(fsys); force(boolptr,+0256);
3069          gen1(op_loc,s.orig); gensp(ASS);
3070        end
3071    end;

3073    procedure statement; {fsys: sos}
3074    var lip:ip; llp:lp; ley:symbol;
3075    begin
3076      assert [labelsy..casesy,endsy] <= fsys;
3077      assert [ident,intest] * fsys = [];
3078      if find2([intest],fsys+[ident],+0257) then
3079        begin llp:=searchlab(b.lobnin,val);
3080          if llp=nil then errint(+0258,val) else
3081            begin if llp^.seen then errint(+0259,val) else llp^.seen:=true;
3082              genlb(llp^.labname)
3083            end;
3084        insym; nextif(colon1,+0260);
3085      end;
3086    if find2([ident,beginsy..casesy],fsys,+0261) then
3087      begin if giveline then if sy<>whilesy then genlin;
3088        if sy=ident then
3089          if id='assert  ' then
3090            begin insym; assertion(fsys) end
3091          else
3092            begin lip:=searchid([vars,field,func,proc]); insym;
3093              if lip^.klass=proc then call(fsys,lip)
3094              else assignment(fsys,lip)
3095            end
3096        else
3097          begin lsy:=sy; insym;
3098            case lsy of
3099              beginsy:
3100                begin compoundstatement(fsys,+0262);  {+0263}
3101                  nextif(endsy,+0264)
3102                end;
3103              gotosy:
3104                gotostatement;
3105              ifsy:
3106                ifstatement(fsys);
3107              casesy:
3108                begin casestatement(fsys); nextif(endsy,+0265) end;
3109              whilesy:
3110                whilestatement(fsys);
3111              repeatsy:
3112                repeatstatement(fsys);
3113              forsy:
3114                forstatement(fsys);
3115              withsy:
3116                withstatement(fsys);
3117            end
3118          end
3119      end
3120  end;

3122  {*************************************************************}

3124  procedure body(fsys:sos; fip:ip);
3125  var i,as,letdlb,namdlb,inidlb:integer; llp:lp;
3126  begin with b do begin namdlb:=0;
3127  {produce PRO}
3128    genpsam(ps_pro,fip); gencst(fip^.headlc);
3129    gencst(ord(fip^.pfpos.lv<=1));
3130  {initialize files}
3131    if level=1 then  {body for main}
3132      begin dlbno:=dlbno+1; inidlb:=dlbno; gend(ps_fwa,inidlb);
3133        gen1(op_mrk,0); gend(op_lae,dlbno); gen1(op_lae,0); gensp(INI);
3134      end;
3135    trace('procentr',fip,namdlb);
3136    dlbno:=dlbno+1; letdlb:=dlbno;
```

```
3137      gen0(ps_fwc,letdlb); gen0(op_beg,letdlb);
3138    {the body itself}
3139      lcmax:=lc; currproc:=fip;
3140      compoundstatement(fsys,+0266);  {+0267}
3141      lcmax:=address(lcmax,0,false);  {align lcmax}
3142      trace('procexit',fip,namdlb);
3143    {undefined or global labels}
3144      llp:=lchain;
3145      while llp<>nil do
3146        begin if not llp^.seen then errint(+0268,llp^.labval);
3147          if llp^.labdlb<>0 then
3148            begin gen1lb(llp^.labdlb); genpnam(ps_rom,fip);
3149              genlb(llp^.labname); genost(lcmax); genend;
3150              {this doesn't work if local generators are around}
3151            end;
3152          llp:=llp^.nextlp
3153        end;
3154    {define BUG size}
3155      gen0(ps_let,letdlb); genost(lcmax-fip^.headlc);
3156    {finish and close files}
3157      treewalk(top^.fname);
3158      if level=1 then
3159        begin gen1lb(inidlb); gen1(ps_con,argc+1);
3160          for i:=0 to argc do with argv[i] do
3161            begin genost(ad);
3162              if (ad=-1) and (i>1) then errid(+0269,name)
3163            end;
3164          genend; gen1(op_mrk,0); gen1(op_loc,0); gensp(HLT)
3165        end
3166      else
3167        begin
3168          if fip^.klass<>func then sz:=0 else
3169            begin
3170              if not (assigned in fip^.iflag) then
3171                errid(-(+0270),fip^.name);
3172              sz:=even(sizeof(fip^.idtype)); push(local,fip^.pfpos.ad,sz);
3173            end;
3174          gen1(op_ret,sz); gen0(ps_end);
3175        end
3176    end end;

3177    {*****************************************************************}

3180    procedure block;  {forward declared}
3181    var ad:integer;
3182    begin with b do begin
3183      assert [labelsy..withsy] <= fsys;
3184      assert [ident,intcst,casesy,endsy,period] * fsys = [];
3185      if find3(labelsy,fsys,+0271) then labeldeclaration(fsys);
3186      if find3(constsy,fsys,+0272) then constdefinition(fsys);
3187      if find3(typesy,fsys,+0273) then typedefinition(fsys);
3188      if find3(varsy,fsys,+0274) then vardeclaration(fsys);
3189      if fl=progp then
3190        begin
3191          if iop[true]<>nil then
3192            begin ad:=address(lc,fhsize+buffsize,false);
3193              argv[1].ad:=ad; iop[true]^.vpos.ad:=ad
3194            end;
3195          if iop[false]<>nil then
3196            begin ad:=address(lc,fhsize+buffsize,false);
3197              argv[0].ad:=ad; iop[false]^.vpos.ad:=ad
3198            end;
3199          if address(lc,0,false)<>0 then gen1(ps_hol,lc);  {align lc}
3200          lc:=progp^.headlc; level:=1
3201        end;  {externals are also extern for the main body}
3202      while find2([procsy,funcsy],fsys,+0275) do pfdeclaration(fsys);
3203      if forwcount<>0 then error(+0276);  {forw proc not specified}
3204      nextif(beginsy,+0277);
3205      body(fsys+[casesy,endsy],fip);
3206      nextif(endsy,+0278);
3207    end end;

3209    {*****************************************************************}

3211    procedure programme(fsys:sos);
3212    var stdin,stdout:boolean; p:ip;
3213    begin
3214      nextif(progsy,+0279); nextif(ident,+0280);
3215      if find3(lparent,fsys+[semicolon],+0281) then
3216        begin
3217          repeat
3218            if sy<>ident then error(+0282) else
3219              begin stdin:=id='input   '; stdout:=id='output  ';
3220                if stdin or stdout then
3221                  begin p:=newip(vars,id,textptr,nil);
3222                    enterid(p); iop[stdout]:=p;
3223                  end
3224                else
3225                  if argc<maxargc then
3226                    begin argc:=argc+1;
3227                      argv[argc].name:=id; argv[argc].ad:=-1
3228                    end;
3229                insym
3230              end
3231          until endofloop(fsys+[rparent,semicolon],
3232                          [ident],comma,+0283);  {+0284}
3233          if argc>maxargc then
3234            begin error(+0285); argc:=maxargc end;
3235          nextif(rparent,+0286);
3236        end;
3237      nextif(semicolon,+0287);
3238      block(fsys,progp);
3239      if opt['1']<>off then
3240        begin gen1(ps_mes,meslino); genost(e.orig); genend end;
3241      eofexpected:=true; nextif(period,+0288);
3242    end;

3244    procedure compile;
3245    var lsys:sos;
3246    begin lsys:=[progsy,labelsy..withsy];
3247      repeat eofexpected:=false;
3248        main:=find2([progsy,labelsy,beginsy..withsy],lsys,+0289);
```



```
3249        if main then programme(lsys) else with b do
3250          begin
3251            if find3(constsy,lsys,+0290) then constdefinition(lsys);
3252            if find3(typesy,lsys,+0291) then typedefinition(lsys);
3253            if find3(varsy,lsys,+0292) then vardeclaration(lsys);
3254            gen1(ps_hol,address(lc,0,false)); lc:=0; level:=1;
3255            while find2([procsy,funcsy],lsys,+0293) do pfdeclaration(lsys);
3256          end;
3257        error(+0294);
3258      until false;  { the only way out is the halt in nextln on eof }
3259    end;

3261    {*****************************************************************}

3263    begin  {main body of pcompiler}
3264      rewrite(errors);
3265      init1; init2; init3; init4;
3266          {all this initializing must be independent of opts}
3267      reset(em1); if not eof(em1) then options(false);
3268      rewrite(em1); write(em1,MAGICLOW,MAGICHIGH);
3269    #ifdef GETREQUIRED
3270      get(input);
3271    #endif
3272      if eof(input) then gen0(ps_eof) else
3273        begin nextch; insym;
3274          handleopts; {initialize all opt dependent stuff}
3275          compile
3276        end;
3277    #ifdef STANDARD
3278    9999: ;
3279    #endif
3280    end.  {pcompiler}
```

```
1     /* collection of options, selected by including or excluding 'defines' */

3     /* select only one of the following: */
4     #       define  V7              1       /* Unix version 7 */
5     /* #    define  V6              1       /* Unix version 6 */
6     /* #    define  V7PLUS          1       /* Unix version 6 plus diff listing */

8     /* select only one of the following: */
9     #       define  C7              1       /* version 7 C-compiler */
10    /* #    define  C6              1       /* version 6 C-compiler */
11    /* #    define  MC6             1       /* something between C6 and C7 */

13    #ifdef BOOT
14    #       define  INT_ONLY        1
15    #endif

17    #ifndef BOOT
18    #       define  HARDWARE_FP     1       /* if you've hardware floating point */
19    /* #    define  INT_ONLY        1       /* for interpreted programs only */
20    #       define  SFLOAT          1       /* for single precision floats */
21    #endif

23    /* Version number of the EM1 object code */
24    #       define  VERSION         2       /* 16 bits number */
```

```
#define sp_fmnem 1
#define sp_mnnem 149
#define sp_fpseu 150
#define sp_npseu 30
#define sp_fil0 180
#define sp_nil0 60
#define sp_fcst0 0
#define sp_ncst0 240
#define sp_il1 240
#define sp_il2 241
#define sp_dl1 242
#define sp_dl2 243
#define sp_dnam 244
#define sp_pnam 245
#define sp_scon 246
#define sp_rcon 247
#define sp_cst1 248
#define sp_cstm 249
#define sp_cst2 250
#define sp_lcon 251
#define sp_cend 255

#define ps_bas 150
#define ps_con 151
#define ps_end 152
#define ps_eof 153
#define ps_exc 154
#define ps_exd 155
#define ps_fwa 156
#define ps_fwc 157
#define ps_fwp 158
#define ps_hol 159
#define ps_ima 160
#define ps_imc 161
#define ps_let 162
#define ps_mes 163
#define ps_pro 164
#define ps_rom 165
#define sp_lpseu 165

#define op_asr 1
#define op_aas 2
#define op_add 3
#define op_adi 4
#define op_and 5
#define op_ans 6
#define op_bag 7
#define op_beq 8
#define op_bes 9
#define op_bge 10
#define op_bgt 11
#define op_ble 12
#define op_blm 13
#define op_bls 14
#define op_blt 15
#define op_bne 16
#define op_brb 17
#define op_brf 18
#define op_cal 19
#define op_cas 20
#define op_cdi 21
#define op_cdf 22
#define op_cfd 23
#define op_cfi 24
#define op_cid 25
#define op_cif 26
#define op_cmd 27
#define op_cmf 28
#define op_cmi 29
#define op_cmp 30
#define op_cms 31
#define op_cmu 32
#define op_com 33
#define op_cos 34
#define op_csa 35
#define op_csb 36
#define op_dad 37
#define op_ddv 38
#define op_dec 39
#define op_dee 40
#define op_del 41
#define op_div 42
#define op_dmd 43
#define op_dmu 44
#define op_dsb 45
#define op_dup 46
#define op_dus 47
#define op_exg 48
#define op_fad 49
#define op_fdv 50
#define op_fef 51
#define op_fif 52
#define op_fmu 53
#define op_fsb 54
#define op_hlt 55
#define op_inc 56
#define op_ine 57
#define op_ini 58
#define op_inn 59
#define op_ins 60
#define op_ior 61
#define op_ios 62
#define op_lab 63
#define op_lae 64
#define op_lai 65
#define op_lal 66
#define op_lar 67
#define op_las 68
#define op_lde 69
#define op_ldf 70
#define op_ldi 71
#define op_lex 72
#define op_lin 73
#define op_lnc 74
#define op_lni 75
#define op_loc 76
#define op_loe 77
#define op_lof 78
#define op_loi 79
#define op_lol 80
#define op_lop 81
#define op_lor 82
#define op_los 83
#define op_lsa 84
#define op_mod 85
#define op_mon 86
#define op_mrk 87
#define op_mrs 88
#define op_mrx 89
#define op_mul 90
#define op_mxs 91
#define op_neg 92
#define op_nop 93
#define op_nul 94
#define op_pad 95
#define op_pab 96
#define op_rck 97
#define op_rcs 98
#define op_res 99
#define op_ret 100
#define op_rol 101
#define op_ror 102
#define op_rtt 103
#define op_sai 104
#define op_sar 105
#define op_sas 106
#define op_sde 107
#define op_sdf 108
#define op_sdl 109
#define op_ses 110
#define op_set 111
#define op_shl 112
#define op_shr 113
#define op_sig 114
#define op_ste 115
#define op_stf 116
#define op_sti 117
#define op_stl 118
#define op_stp 119
#define op_str 120
#define op_sts 121
#define op_sub 122
#define op_teq 123
#define op_tge 124
#define op_tgt 125
#define op_tle 126
#define op_tlt 127
#define op_tne 128
#define op_trp 129
#define op_xor 130
#define op_xos 131
#define op_zeq 132
#define op_zge 133
#define op_zgt 134
#define op_zle 135
#define op_zlt 136
#define op_zne 137
#define op_zre 138
#define op_zrl 139
#define sp_lmnem 139
```

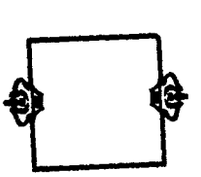


```
non-standard feature used
identifier '$s' declared twice
end of file encountered
bad line directive
unsigned real: digit of fraction expected
unsigned real: digit of exponent expected
unsigned real: too many digits (>72)
unsigned integer: too many digits (>72)
unsigned integer: overflow (>32767)
string constant: must not exceed one line
string constant: at least one character expected
string constant: double quotes not allowed (see o option)
string constant: too long (>72 chars)
comment: ';' seen (statements skipped?)
bad character
identifier '$s' not declared
location counter overflow: arrays too big
arraysize too big
variable '$s' never used
variable '$s' never assigned
the files contained in '$s' are not closed automatically
constant expected
constant: only integers and reals may be signed
constant: out of bounds
simple type expected
enumerated type: element identifier expected
enumerated type: ',' or ')' expected
enumerated type: ',' expected
enumerated type: ')' expected
subrange type: type must be scalar, but not real
subrange type: '..' expected
subrange type: type of lower and upper bound incompatible
subrange type: lower bound exceeds upper bound
array type: '[' expected
conformant array: low bound identifier expected
conformant array: '..' expected
conformant array: high bound identifier expected
conformant array: ':' expected
conformant array: index type identifier expected
array type: index type not bounded
array type: index separator or ']' expected
array type: index separator expected
array type: ']' expected
array type: 'of' expected
record variant part: tag type identifier expected
record variant part: tag type identifier expected
record variant part: type must be bounded
record variant part: 'of' expected
record variant: type of case label and tag incompatible
record variant: multiple defined case label
record variant: ',' or ':' expected
record variant: ',' expected
record variant: ':' expected
record variant: '(' expected
record variant: ')' expected
record variant part: ';' or end of variant list expected
record variant part: ';' expected
record variant part: end of variant list expected
record variant part: there must be a variant for each tag value
field list: record section expected
record section: field identifier expected
record section: ',' or ':' expected
record section: ',' expected
record section: ':' expected
field list: ';' or end of record section list expected
field list: ';' expected
field list: end of record section list expected
type expected
type: simple and pointer type may not be packed
pointer type: type identifier expected
pointer type: type identifier expected
record type: 'end' expected
set type: 'of' expected
set of integer: the i option dictates the number of bits (default 16)
set type: base type not bounded
set type: too many elements in set (see i option)
file type: 'of' expected
file type: files within files not allowed
var parameter: type identifier or conformant array expected
var parameter: type identifier expected
label declaration: unsigned integer expected
label declaration: label '$i' multiple declared
label declaration: ',' or ':' expected
label declaration: ',' expected
label declaration: ';' expected
const declaration: constant identifier expected
const declaration: '=' expected
const declaration: ';' expected
const declaration: constant identifier or 'type', 'var', 'procedure', 'function' or
type declaration: type identifier expected
type declaration: '=' expected
type declaration: ';' expected
type declaration: type identifier or 'var', 'procedure', 'function' or 'begin' expect
var declaration: var identifier expected
var declaration: ',' or ':' expected
var declaration: ',' expected
var declaration: ';' expected
var declaration: ';' expected
var declaration: var identifier or 'procedure', 'function' or 'begin' expected
parameter list: 'var','procedure','function' or identifier expected
parameter list: parameter identifier expected
parameter list: ',' or ':' expected
parameter list: ',' expected
parameter list: ':' expected
parameter list: type identifier expected
parameter list: ';' or ')' expected
parameter list: ';' expected
proc/func declaration: proc/func identifier expected
proc/func declaration: previous declaration of '$s' was not forward
proc/func declaration: parameter list expected
parameterlist: ')' expected
func declaration: ':' expected
```

```
113  func declaration: result type identifier expected
114  func declaration: result type must be scalar, subrange or pointer
115  proc/func declaration: ';' expected
116  proc/func declaration: block or directive expected
117  proc/func declaration: '%s' again forward declared
118  proc/func declaration: '%s' unknown directive
119  proc/func declaration: ';' expected
120  indexed variable: '[' only allowed following array variables
121  indexed variable: index type not compatible with declaration
122  indexed variable: ',' or ']' expected
123  indexed variable: ',' expected
124  assignment: standard function not allowed as destination
125  assignment: cannot store the function result
126  assignment: formal parameter function not allowed as destination
127  assignment: function identifier may not be de-referenced
128  variable: '[', '.', '^' or end of variable expected
129  indexed variable: ']' expected
130  field designator: field identifier expected
131  field designator: '.' only allowed following record variables
132  field designator: no field '%s' in this record
133  referenced variable: '^' not allowed following zero-terminated strings
134  referenced variable: '^' only allowed following pointer or file variables
135  variable: var or field identifier expected
136  call: array parameter not conformable
137  call: type of actual and formal variable parameter not similar
138  call: packed elements not allowed as variable parameter
139  call: type of actual and formal value parameter not compatible
140  call: proc/func identifier expected
141  call: standard proc/func may not be used as parameter
142  call: parameter lists of actual and formal proc/func incompatible
143  call: ',' or ')' expected
144  call: too many actual parameters supplied
145  call: ')' expected
146  call: too few actual parameters supplied
147  read(ln): type must be integer, char or real
148  write(ln): type must be integer, char, real, string or boolean
149  write(ln): ':', ',' or ')' expected
150  write(ln): field width must be integer
151  write(ln): ':', ',' or ')' expected
152  write(ln): precision must be integer
153  write(ln): precision may only be specified for reals
154  read/write: too few actual parameters supplied
155  read/write: standard input/output not mentioned in program heading
156  read/write: ',' or ')' expected
157  read/write: type of parameter not the same as that of the file elements
158  read/write: parameter list expected
159  readln/writeln: standard input/output not mentioned in program heading
160  readln/writeln: only allowed on text files
161  eof/eoln/page: file variable expected
162  eoln/page: text file variable expected
163  eof/eoln/page: standard input/output not mentioned in program heading
164  new/dispose: pointer variable expected
165  new/dispose: C-type strings not allowed here
166  new/dispose: ',' or ')' expected
167  new/dispose: too many actual parameters supplied
168  new/dispose: type of tagfield value is incompatible with declaration
169  put/get: file variable expected
170  reset/rewrite: file variable expected
171  mark/release: pointer variable expected
172  pack/unpack: array types are incompatible
173  pack/unpack: only for arrays
174  call: '(' or end of call expected
175  standard proc/func: parameter list expected
176  standard proc/func: parameter type incompatible with specification
177  pack: ',' expected
178  pack: ',' expected
179  unpack: ',' expected
180  unpack: ',' expected
181  halt: integer expected
182  abs: integer or real expected
183  sqr: integer or real expected
184  ord: type must be scalar or subrange, but not real
185  pred/succ: type must be scalar or subrange, but not real
186  trunc: real argument required
187  round: real argument required
188  call: ')' expected
189  expression: left and right operand are incompatible
190  set: base type must be bounded or of type integer
191  set: base type upper bound exceeds maximum set element number (255)
192  set: incompatible elements
193  set: ']' or element list expected
194  set: '..', ',' or ']' expected
195  set: elements do not fit (see i option)
196  set: ',' or ')' expected
197  set: ',' expected
198  factor expected
199  factor: ')' expected
200  factor: type of factor must be boolean
201  set: ']' expected
202  term: multiplying operator or end of term expected
203  term: '*' only defined for integers, reals and sets
204  term: '/' only defined for integers and reals
205  term: 'div' only defined for integers
206  term: 'mod' only defined for integers
207  term: 'and' only defined for booleans
208  simple expression: only integers and reals may be signed
209  simple expression: adding operator or end of simple expression expected
210  simple expression: '+' only defined for integers, reals and sets
211  simple expression: '-' only defined for integers, reals and sets
212  simple expression: 'or' only defined for booleans
213  expression: relational operator or end of expression expected
214  expression: set expected
215  expression: left operand of 'in' not compatible with base type of right operand
216  expression: only '=' and '<>' allowed on pointers
217  expression: '<' and '>' not allowed on sets
218  expression: comparison of arrays only allowed for strings
219  expression: comparison of records not allowed
220  expression: comparison of files not allowed
221  assignment: ':=' expected
222  assignment: left and right hand side incompatible
223  goto statement: unsigned integer expected
224  goto statement: label '%i' not declared
```



```
225  if statement: type of expression must be boolean
226  if statement: 'then' expected
227  if statement: 'else' or end of if statement expected
228  case statement: type must be scalar or subrange, but not real
229  case statement: 'of' expected
230  case statement: incompatible case label
231  case statement: multiple defined case label
232  case statement: ',' or ':' expected
233  case statement: ',' expected
234  case statement: ':' expected
235  case statement: ';' or 'end' expected
236  case statement: ';' expected
237  case statement: 'end' expected
238  repeat statement: ';' or 'until' expected
239  repeat statement: ';' expected
240  repeat statement: 'until' expected
241  repeat statement: type of expression must be boolean
242  while statement: type of expression must be boolean
243  while statement: 'do' expected
244  for statement: control variable expected
245  for statement: control variable must be local
246  for statement: type must be scalar or subrange, but not real
247  for statement: ':=' expected
248  for statement: type of initial value and control variable incompatible
249  for statement: 'to' or 'downto' expected
250  for statement: type of final value and control variable incompatible
251  for statement: 'do' expected
252  with statement: record variable expected
253  with statement: ',' or 'do' expected
254  with statement: ',' expected
255  with statement: 'do' expected
256  assertion: type of expression must be boolean
257  statement expected
258  label '%i' not declared
259  label '%i' multiple defined
260  statement: ';' expected
261  unlabeled statement expected
262  compound statement: ';' or 'end' expected
263  compound statement: ';' expected
264  compound statement: 'end' expected
265  case statement: 'end' expected
266  body: ';' or 'end' expected
267  body: ';' expected
268  body: label '%i' declared, but never defined
269  program parameter '%s' not declared
270  function '%s' never assigned
271  block: declaration or body expected
272  block: 'const', 'type', 'var', 'procedure', 'function' or 'begin' expected
273  block: 'type', 'var', 'procedure', 'function' or 'begin' expected
274  block: 'var', 'procedure', 'function' or 'begin' expected
275  block: 'procedure', 'function' or 'begin' expected
276  block: unsatisfied forward proc/func declaration(s)
277  block: 'begin' expected
278  block: 'end' expected
279  program heading: 'program' expected
280  program heading: program identifier expected
281  program heading: file identifier list expected
282  program heading: file identifier expected
283  program heading: ',' or ')' expected
284  program heading: ',' expected
285  program heading: maximum number of file arguments exceeded (12)
286  program heading: ')' expected
287  program heading: ';' expected
288  program: ',' expected
289  'program' expected
290  module: 'const', 'type', 'var', 'procedure' or 'function' expected
291  module: 'type', 'var', 'procedure' or 'function' expected
292  module: 'var', 'procedure' or 'function' expected
293  module: 'procedure' or 'function' expected
294  garbage at end of program
```



```
{**
*         OPTIONS - RETURN CONTROL STATEMENT OPTION SETTING.
*         COPYRIGHT (C) UNIVERSITY OF MINNESOTA - 1978.
*         A. B. NICKEL.          77/06/02.
*
*         THE ORIGINAL ROUTINE -OPTION- ACCEPTED A ONE-CHARACTER
*         OPTION NAME AND RETURNED AN OPTION SETTING OF +, -, =,
*         OR A POSITIVE INTEGER.
*
*         THIS VERSION, CALLED -OPTIONS-, ACCEPTS ANY STRING
*         OF 1 TO 10 ALPHANUMERIC CHARACTERS (STARTING WITH AN
*         ALPHA) AS THE OPTION NAME AND RETURNS A STRING OF
*         1 TO 10 CHARACTERS OR A POSITIVE INTEGER AS THE OPTION
*         SETTING. AN EQUALS SIGN MAY BE USED BETWEEN AN
*         OPTION NAME AND ITS OPTION SETTING. IF THERE IS NO
*         OPTION SETTING AFTER THE EQUALS SIGN, THEN THE
*         EQUALS SIGN ITSELF IS USED AS THE OPTION SETTING. IF
*         THE OPTION NAME IS FOLLOWED BY A COMMA, PERIOD, OR
*         RIGHT PARENTHESIS, THE OPTION SETTING IS RETURNED AS A
*         STRING OF 10 BLANK CHARACTERS.
*
*         THE INPUT VARIABLE -NAME- IS NOW TYPE ALFA, AND IN THE
*         RECORD TYPE -SETTING-, THE FIELD -ONOFF- IS NOW TYPE
*         ALFA.
*
*         SEE THE PASCLIB WRITEUP FOR EXTERNAL DOCUMENTATION.
*         NOTE THAT THE NAME OF THIS VERSION IS -OPTIONS-.
*
*         SPIKE LEONARD - SANDIA NATIONAL LABORATORIES, LIVERMORE
*         24 FEB 1981
*)

FUNCTION OPTIONS(NAME: ALFA; VAR S: SETTING): BOOLEAN;

CONST
   CSADDRESS = 788 (*CONTROL STATEMENT ADDRESS*);
TYPE
   CSIMAGEP = RECORD CASE BOOLEAN OF
                FALSE: (I: INTEGER);
                TRUE: (P: ^LONGCORE)
              END;
   LONGCORE = PACKED ARRAY[1..80] OF CHAR;
VAR
   CSIMAGE: CSIMAGEP;
   OPNAME: ALFA;
   I: INTEGER                    (* INDEX IN CSIMAGE *);
   J: INTEGER                    (* INDEX FOR OPNAME *);
   K: INTEGER                    (* INDEX FOR S.ONOFF *);
   FOUND: BOOLEAN;

BEGIN (*OPTIONS*)
   FOUND := FALSE;
   S.SWITCH := FALSE;  S.SIZE := 0;
   CSIMAGE.A := CSADDRESS;
   I := 1 (*SKIP PROGRAM NAME AND PARAMETERS.*);
      WHILE CSIMAGE.P^[I] IN [' ','A'..'Z', '0'..'9', '.', '='] DO
         I := I + 1;
      IF NOT (CSIMAGE.P^[I] IN ['('], ',']) THEN
         I := I + 1 (*SKIP SLASH IF FIRST DELIMITER.*);
      WHILE NOT (CSIMAGE.P^[I] IN ['/', '('], ',']) DO
         I := I + 1;
   IF CSIMAGE.P^[I] = '/' THEN (*CRACK OPTIONS.*)
     REPEAT
       I := I + 1;
       J := 1;
       OPNAME := '          ';
       IF CSIMAGE.P^[I] IN ['A'..'Z'] THEN BEGIN
         WHILE (CSIMAGE.P^[I] IN ['A'..'Z','0'..'9']) AND NOT FOUND DO
           BEGIN
           OPNAME[J] := CSIMAGE.P^[I];
           J := J + 1;
           I := I + 1;
           IF (NAME = OPNAME) AND NOT (CSIMAGE.P^[I] IN ['A'..'Z'])
             THEN BEGIN
             FOUND := TRUE;
             IF (CSIMAGE.P^[I] = '=') AND
               NOT (CSIMAGE.P^[I+1] IN ['(',',','.','=',')']) THEN
                 I := I + 1;
             S.SWITCH := NOT (CSIMAGE.P^[I] IN ['0'..'9']);
             IF S.SWITCH THEN BEGIN
               S.ONOFF := '          ';
               K := 1;
               WHILE NOT (CSIMAGE.P^[I] IN ['(',',','.','=',')']) DO BEGIN
                 S.ONOFF[K] := CSIMAGE.P^[I];
                 K := K + 1;
                 I := I + 1;
                 END;
               END
             ELSE
               WHILE CSIMAGE.P^[I] IN ['0'..'9'] DO BEGIN
                 S.SIZE := S.SIZE*10
                   + (ORD(CSIMAGE.P^[I]) - ORD('0'));
                 I := I + 1;
                 END;
             END;
           END;
         END;
       IF NOT FOUND THEN
         WHILE NOT (CSIMAGE.P^[I] IN ['(',',','.','=',')']) DO I := I + 1;
     UNTIL (CSIMAGE.P^[I] IN ['(',',','.',')']) OR FOUND;
   OPTIONS := FOUND;
END (*OPTIONS*);
```


TREEPRINT - A Package to Print Trees on any Character Printer

Med Fread
Kevin Carosso

Mathematics Department
Harvey Mudd College
Claremont, Calif. 91711

One of the problems facing a programmer who deals with complex linked data structures in Pascal is the inability to display such a structure in a graphical form. Usually it is too much to ask a system debugging tool to even understand records and pointers, let alone display a structure using them in the way it would appear in a good textbook. Likewise very few operating systems have a package of routines to display structures automatically. Pascal has a tremendous advantage over many languages in its ability to support definable types and structures. If the environment is incapable of dealing with these features, they become far less useful.

This lack became apparent to us in the process of writing an algebraic expression parser which produced internal N-ary trees. There was no way at the time under our operating system debugger (VAX/VMS) to get at the data structure we were generating. When the routines produced an incorrect tree we had no way of finding the specific error.

Our frustration led to the development of TREEPRINT. Starting with the algorithm of Jean Vaucher [1], we designed a general-purpose tool capable of displaying any N-ary tree on any character output device. The trees are displayed in a pleasant visual form and in the manner in which they would appear if drawn by hand. We feel that TREEPRINT is of general use -- hence its presentation here.

The structure of TREEPRINT is that of an independent collection of subroutines that any program can call. Unfortunately standard Pascal does not support this form, while our Pascal environment does. However, building TREEPRINT directly into a program should present no difficulty.

TREEPRINT requires no knowledge of the format of the data structure it is printing. It has even been used to print a tabular linked structure within a FORTRAN program! In order to allow this, two procedures are passed in the call to TREEPRINT. One is used to "walk" the tree, the other to print identifying labels for a given node. Other parameters are values such as the size of the nodes, the width of the page, etc. One of the advantages of this calling mechanism is that a single version of TREEPRINT can be used to display wildly different structures, even when they are within the same program.

One of the major features of TREEPRINT is its ability to span pages. A tree that is too wide to fit on one page is printed out in "stripes" which are taped together edge-to-edge after printing. In addition trees may optionally be printed either upside-down or reversed from left-to-right.

The method used by TREEPRINT is detailed in Vaucher's work [1]. In its current implementation additional support for N-ary structures has been added, as well as full connecting-arc printing and the reversal features. Basically, TREEPRINT walks the input tree and constructs an analogous structure of its own which indicates the positions of every node. The new structure is linked along the left edge and across the page from left-to-right. Once this structure is completed, TREEPRINT walks the new structures and prints it out in order. Once printout is finished, the generated structure is DISPOSE'd of.

There are only two minor problems in TREEPRINT currently. The first is that a structure which contains circular loops will hang the routine. This could be detected in the POSITION phase of TREEPRINT by checking each new node against all of its ancestors. However, if used in a non-Pascal application, this might fail due to problems in comparing pointers. If this check is necessary we suggest it be implemented in the LOWERNODE procedure passed to TREEPRINT. This procedure at least understands the type of pointer it is dealing with.

The second problem is a feature of the POSITION routine which centers a node above its sons. This tends to make the trees generated wider than necessary. This is largely a matter of taste -- some minor changes would remove this.

The listing of TREEPRINT which follows should serve to document the method of calling the routine. The functions of the user-supplied procedures are also detailed.

References

[1] Vaucher, Jean, "Pretty-Printing of Trees." Software- Practice and Experience, Vol. 10, pp. 553-561 (1980).

[2] Myers, Brad, Displaying Data Structures for Interactive Debugging, Palo Alto: Xerox PARC CSL-80-7 (1980).

[3] Sweet, Richard, Empirical Estimates of Program Entropy, Appendix B - "Implementation description", Palo Alto: Xerox PARC CSL-78-3 (1978).

```
module TREEPRINT (input,output);

(*
    TREEPRINT - A routine to print N-ary trees on any character
      printer. This routine takes as input an arbitrary N-ary tree,
      some interface routines, and assorted printer parameters and
      writes a pictorial representation of that tree to a file. The
      tree is nicely formatted and is divided into vertical stripes
      that can be taped together after printing. Options exist to
      print the tree backwards or upside down if desired.

      The algorithm for TREEPRINT originally appeared in "Pretty-
      Printing of Trees", by Jean G. Vaucher, Software-Practice and
      Experience, Vol. 10, 553-561 (1980). The algorithm used here
      has been modified to support N-ary tree structures and to have
      more sophisticated printer format control. Aside from a common
      method of constructing an ancillary data structure and some
      variable names, they are now very dissimiliar.

      TREEPRINT was written by Ned Freed and Kevin Carosso,
      5-Feb-81. It may be freely distributed, copied and modified
      provided that this note and the above reference are included.
      TREEPRINT may not be distributed for any fee other than cost
      of duplication.

    INPUT - The call to TREEPRINT is:
        TREEPRINT (TREE,TREEFILE,PAGESIZE,VERTKEYLENGTH,
                   HORIKEYLENGTH,PRINTKEY,LOWERNODE)

        where the parameters are:

          TREE - The root of the tree to be printed. The nodes of
                 the tree are of arbitrary type, as TREEPRINT
                 does not read them itself but calls procedure
                 LOWERNODE to do so. In a modular enviroment
                 this should present no problems. If TREEPRINT
                 is to be installed directly in a program TREE
                 will have to be changed to agree in type with
                 the actual tree's nodes.
          TREEFILE - A file variable of type text. The tree is
                 written into this file.
          PAGESIZE - The size of the page on output represented
                 as an integer count of the number of available
                 columns. The maximum page size is 512. Any size
                 greater than 512 will be changed to 512.
          LOWERNODE - A user procedure TREEPRINT calls to walk
                 the user's tree. The format for the call is
                 described below along with the functions
                 LOWERNODE must perform.
          PRINTKEY - A user procedure TREEPRINT calls to print
                 out a single line of a keyword description of
                 some node in the user's tree. The description
                 may be multi-line and of any width. The call
                 format is described below.
          VERTKEYLENGTH - The number of lines of a description
                 printed by PRINTKEY. This must be a constant
                 over all nodes. If VERTKEYLENGTH is negative,
                 its absolute value is used as the key length and
                 the whole tree is inverted on the vertical axis.
          HORIKEYLENGTH - The number of characters in a single
                 line of a description printed by PRINTKEY. This
                 must be a constant. If negative the absolute
                 value of HORIKEYLENGTH is used and the whole
                 tree is inverted from left to right.

    CALLS TO USER PROCEDURES - The calls to user-supplied procedures
      have the following format and function:

        PRINTKEY (LINENUMBER,LINELENGTH,NODE)
          LINENUMBER - The line of the node description to print.
            This varies from 1 to VERTKEYLENGTH. Since TREEPRINT
            operates on a line-at-a-time basis, PRINTKEY must be
            able to break up the output in a similiar fashion.
          LINELENGTH - The length of the line. PRINTKEY must
            output this many characters to TREEFILE - no more, no
            less.
          NODE - The node of the user's tree to derive information
            from.

        LOWERNODE (NODE,SONNUMBER)
          SONNUMBER - The sub-node to return. A general N-ary tree
            will have N of them.
          NODE - The node of the user's tree to derive the
            information from.
          LOWERNODE, on return should equal NIL if that node does
            not exist, NODE if the SONNUMBER is illegal, and
            otherwise a valid sub-node. Note that circular
            structures will hang treeprint thoroughly. The condition
            that LOWERNODE returns NODE when N is exceeded must be
            strictly adhered to, as TREEPRINT uses this to know
            where to stop. LOWERNODE is used to hide the interface
            between TREEPRINT and the user's tree so that no format
            details of the tree need be resident in TREEPRINT.

      OUTPUT - All output is directed to TREEFILE. There are no error
        conditions or messages.
*)

(* The declaration of the user's node type. If type checking is a
   problem this should be changed to match the type for the actual
   nodes in a tree. *)

type
    nodeptr = ^integer;

procedure treeprint (tree : nodeptr; var treefile : text;
                     pagesize, vertkeylength, horikeylength :
                       integer; procedure printkey; function
                       lowernode : nodeptr);


type

    reflink = ^link;
    link = record
             next : reflink;
             pnode : nodeptr;
             pos : integer;
             lstem : boolean;
             ustem : boolean;
```



```
          end;

    refhead = ^head;
    head = record
             next : refhead;
             first : reflink;
           end;


var
    maxposition, minposition, width, w, charp : integer;
    startposition, beginposition, endposition : integer;
    pagewidth, p, i, j, stemlength, vertnodelength : integer;
    endloop : boolean;
    line : packed array [1..512] of char;
    L, oldL : reflink;
    lines, alines, H, D : refhead;

    procedure cout (c : char);

        (* Cout places a character in the line buffer at the
           current character position. The pointer charp is
           incremented by this action to reflect the change. *)


    begin  (* Cout *)
        charp := charp + 1;
        line[charp] := c;
    end;  (* Cout *)

    procedure cdump;

        (* Cdump dumps all characters that have accumulated in
           the line buffer. No characters are omitted and no
           cr-lf is appended. *)

    begin  (* Cdump *)
        if charp > 0 then for charp := 1 to charp do
          write (treefile,line[charp]);
        charp := 0;
    end;  (* Cdump *)

    procedure ctrim;

        (* Ctrim dumps all characters that have accumulated in
           the line buffer with trailing spaces removed. A
           WRITELN is used to end the line. *)


    begin  (* Ctrim *)
        while (charp > 0) and (line[charp] = ' ') do
          charp := charp - 1;
        if charp > 0 then for charp := 1 to charp do
          write (treefile,line[charp]);
        charp := 0;
        writeln (treefile);
    end;  (* Ctrim *)

    function position (N : nodeptr; var H : refhead; pos : integer)
                      : reflink;


        (* Position is a recursive function that positions all the
           nodes of the tree on the print page. In doing so, it
           constructs an auxiliary data structure that is connected
           by line number along the edge and position from left to
           right. In addition, it stores some of the original tree
           connections for arc printing. *)


    var
        over, lastover, nodecount : integer;
        Nlower : nodeptr;
        L, left, right : reflink;
        needright : boolean;

    begin  (* Position *)
        if N = nil then  (* Be defensive about illegal nodes. *)
        position := nil
        else
        begin  (* Create a new node in our tree. *)
            new (L);
            position := L;
            L^.pnode := N;
            L^.ustem := false;
            if H = nil then
            begin  (* A new line has been reached. *)
                new (H);
                H^.next := nil;
                L^.next := nil;
            end
            else
            begin  (* Shift position if conflicting. *)
                L^.next := H^.first;
                if H^.first^.pos < pos + 2 then
                  pos := H^.first^.pos - 2;
            end;
            H^.first := L;
            nodecount := 0;
            over := 1;
            repeat  (* Count the number of lower nodes. *)
                Nlower := lowernode (N,over);
                if ((Nlower <> N) and (Nlower <> nil)) then
                  nodecount := nodecount + 1;
                over := over + 1;
            until Nlower = nil;
            if nodecount > 0 then
            begin  (* There are lower nodes, loop to position. *)
                L^.lstem := true;
                lastover := nodecount - 1;
                nodecount := over;
                over := - lastover;
                needright := true;
                repeat  (* Recursively evaluate lower positions. *)
                    repeat  (* Find one that is non-nil. *)
                        if nodecount > 0 then
                          Nlower := lowernode (N,nodecount)
                        else
                          Nlower := N;
                        nodecount := nodecount - 1;
                    until Nlower <> nil;
                    if Nlower <> N then
                    begin
                        left :=
                          position (Nlower, H^.next, pos + over);
```

```
240                         if needright then
241                         begin
242                             right := left;
243                             needright := false;
244                         end
245                         else left^.ustem := true;
246                         over := over + 2;
247                     end;
248                 until (over > lastover) or (nodecount <= 0);
249                 pos := (left^.pos + right^.pos) div 2;
250             end
251             else
252                 L^.lstem := false;
253             if pos > maxposition then maxposition := pos
254             else
255                 if pos < minposition then minposition := pos;
256             L^.pos := pos;
257         end;  (* if H = nil *)
258     end;  (* Position *)
259
260 begin  (* Treeprint *)
261
262     (* Initialize various variables. *)
263
264     lines := nil;
265     minposition := 0;
266     maxposition := 0;
267     charp := 0;
268
269     (* Do various width and length calculations. *)
270
271     if pagesize > 512 then pagesize := 512;
272     width := abs (horikeylength) + 4;
273     stemlength := abs (vertkeylength) + 1;
274     vertnodelength := 3 * abs (vertkeylength) + 4;
275     if (width mod 2) = 0 then width := width + 1;
276     pagewidth := pagesize div width;
277
278     (* Construct our data structure and compute positions. *)
279
280     oldL := position (tree,lines,0);
281
282     (* If the horizontal reverse option is selected, reverse
283        every node on every line of the data structure. It is
284        also necessary to switch around the states of the USTEM
285        flags that tell who connects above a given node. *)
286
287     if horikeylength < 0 then
288     begin
289         H := lines;
290         while H <> nil do
291         begin
292             H^.first^.pos := maxposition -
293                              H^.first^.pos + minposition;
294             if H^.first^.ustem then
295             begin
296                 H^.first^.ustem := false;
297                 endloop := true;
298             end
299             else
300                 endloop := false;
301             L := nil;
302             while H^.first^.next <> nil do
303             begin
304                 H^.first^.next^.pos := maxposition -
305                     H^.first^.next^.pos + minposition;
306                 if H^.first^.next^.ustem then
307                 begin
308                     if not endloop then
309                     begin
310                         H^.first^.next^.ustem := false;
311                         endloop := true;
312                     end;
313                 end
314                 else
315                     if endloop then
316                     begin
317                         H^.first^.next^.ustem := true;
318                         endloop := false;
319                     end;
320                 oldL := H^.first^.next;
321                 H^.first^.next := L;
322                 L := H^.first;
323                 H^.first := oldL;
324             end;
325             H^.first^.next := L;
326             H := H^.next;
327         end;
328     end;
329
330     (* If the vertical reverse option is selected, reverse the
331        entire tree on the vertical axis by flipping all the
332        head nodes along the edge. Arc reversal is handled in
333        the actual arc generation routines. They will scan the
334        previous line of info instead of the current one. *)
335
336     slines := lines;
337     if vertkeylength < 0 then
338     begin
339         H := nil;
340         while lines^.next <> nil do
341         begin
342             D := lines^.next;
343             lines^.next := H;
344             H := lines;
345             lines := D;
346         end;
347         lines^.next := H;
348     end;
349
350     (* Break up entire width into pages and loop over each. *)
351
352     startposition := minposition;
353     while startposition <= maxposition do
354     begin
355         page (treefile);
356         H := lines;
357         while H <> nil do
358         begin  (* Loop over all lines possible. *)
359             oldL := H^.first;
```



```
360             repeat  (* Find a node on current strip. *)
361                 endloop := true;
362                 if oldL <> nil then
363                     if oldL^.pos < startposition then
364                         begin  (* Reject this node. *)
365                             oldL := oldL^.next;
366                             endloop := false;
367                         end;
368             until endloop;
369             for i := 1 to vertnodelength do
370             begin  (* Loop for each print line in a node. *)
371                 L := oldL;
372                 p := startposition;
373                 while (p < startposition + pagewidth) and
374                     (L <> nil) do
375                 begin  (* Scan for nodes we need to draw. *)
376                     if L^.pos = p then
377                     begin  (* Found node at current position. *)
378                         if (i <= stemlength) then
379                         begin  (* Draw upper stem part of node. *)
380                             for w := 1 to (width div 2) do
381                                 cout (' ');
382                             if ((vertkeylength < 0) and L^.lstem)
383                                 or ((vertkeylength >= 0) and
384                                 (H <> slines)) then cout ('*')
385                             else cout (' ');
386                             for w := 1 to (width div 2) do
387                                 cout (' ');
388                         end
389                         else
390                         if (vertnodelength - i) < stemlength then
391                         begin  (* Draw lower stem part of node. *)
392                             for w := 1 to (width div 2) do
393                                 cout (' ');
394                             if ((vertkeylength >= 0) and L^.lstem)
395                                 or ((vertkeylength < 0) and
396                                 (H <> slines)) then cout ('*')
397                             else cout (' ');
398                             for w := 1 to (width div 2) do
399                                 cout (' ');
400                         end
401                         else
402                         if (i >= stemlength + 2)
403                             and (i <= stemlength * 2) then
404                             begin  (* Print node identifier. *)
405                                 cout ('*');
406                                 cout (' ');
407                                 cdump;
408                                 printkey (i - stemlength - 1,
409                                     abs (horikeylength), L^.pnode);
410                                 cout (' ');
411                                 cout ('*');
412                             end
413                         else
414                             for w := 1 to width do cout ('*');
415                         L := L^.next;
416                     end
417                     else
418                         for w := 1 to width do cout (' ');
419                     p := p + 1;
420                 end;
421                 ctrim;
422             end; (* for *)
423
424             (* Select the proper line to obtain arc info from. *)
425
426             if vertkeylength >= 0 then
427             begin
428                 if H^.next <> nil
429                 then L := H^.next^.first
430                 else L := nil;
431             end
432             else L := H^.first;
433
434             p := startposition;
435             while (p < startposition + pagewidth) and (L <> nil) do
436             begin
437                 endposition := L^.pos;
438                 beginposition := L^.pos;
439                 if L^.ustem then
440                     while (L^.next <> nil) and L^.ustem do
441                     begin
442                         L := L^.next;
443                         endposition := L^.pos;
444                     end;
445                 L := L^.next;
446                 if (beginposition < startposition + pagewidth)
447                    and (endposition >= startposition) then
448                 begin  (* Found an arc we should draw. *)
449                     while p < beginposition do
450                     begin  (* Space over to proper position. *)
451                         for w := 1 to width do cout (' ');
452                         p := p + 1;
453                     end;
454                     if beginposition = endposition then
455                     begin  (* Case of one node directly below. *)
456                         for w := 1 to (width div 2) do cout (' ');
457                         if H <> slines then cout('*')
458                         else cout(' ');
459                         for w := 1 to (width div 2) do cout (' ');
460                         p := p + 1;
461                     end
462                     else
463                     begin  (* Normal multi-segment arc, then. *)
464                         if p = beginposition then
465                         begin  (* Begin with a half segment. *)
466                             for w := 1 to (width div 2) do
467                                 cout (' ');
468                             for w := (width div 2) to width-1 do
469                                 cout ('*');
470                             p := p + 1;
471                         end;
472                         while (p < endposition) and
473                             (p < startposition + pagewidth) do
474                         begin  (* Connect to the end segment. *)
475                             for w := 1 to width do cout ('*');
476                             p := p + 1;
477                         end;
478                         if p < startposition + pagewidth then
479                         begin  (* Draw end segment of the arc. *)
```

```
                                for w := (width div 2) to width-1 do
                                    cout ('*');
                                for w := 1 to (width div 2) do
                                    cout (' ');
                                p := p + 1;
                            end;
                        end;
                    end;
                end;
                ctrim;

                (* We have now finished an entire line of tree. *)

                H := H^.next;
            end;  (* while H<>nil *)

            (* Start up on a new page of material. *)

            startposition := startposition + pagewidth;
        end;  (* while startposition <= maxposition *)
        (* All output is finished. It is now time to close out our extra
           data structure. *)
        while lines <> nil do
        begin  (* Collect a line of stuff and dispose. *)
            H := lines^.next;
            while lines^.first <> nil do
            begin  (* Kill a node. *)
                L := lines^.first^.next;
                dispose (lines^.first);
                lines^.first := L;
            end;
            dispose (lines);
            lines := H;
        end;
    end;  (* Treeprint *)

end.  (* Of module TREEPRINT *)
```

****

```
{
    Written by:     Tom Slone
                    Nov 15, 1980
                    at Lehigh University,
                    Bethlehem, PA 18015
                    on a DEC System 20

    (c) Copyright 1980

The author grants permission to copy for non-profit use, providing
this comment remains.
}
PROGRAM compress(in_file, out_file);
{
    This program takes a text file and creates a compressed
version using Huffman codes.  Savings average 30-40%.  The compressed
file can be restored to normal using the sister program called
"RECALL".
}

LABEL
    13;

CONST
    minchar = 1 (This is the ordinal of the smallest character not to
  be ignored, i.e. in this case only nulls are ignored.);
    maxdepth = 64 (This should be 2**n, where n is the number of bits
                   per character in the character set.);
    maxlength = maxdepth;
    maxint = 34359738367;
    bit_size = 36 (Number of bits per machine word);

TYPE
    bit = 0.. 1;
    out_word = PACKED ARRAY [1.. bit_size] OF bit;
    alphabet = minchar .. 127;
    newchar = RECORD
                  length: 0.. maxdepth;
                  nchar: PACKED ARRAY [1.. maxdepth] OF bit
              END;
    treept = ^ tree;
    tree = RECORD
               sum: integer;
               left, right: treept
           END;

VAR
    num_in_chars, num_out_words: integer;
    pos: integer;
    wd: out_word;
    in_file: text;
    out_file: FILE OF out_word;
    tally: ARRAY [alphabet] OF RECORD
                                   marked: boolean;
                                   num_of: integer
                               END;
    trees: RECORD
               t_num: 0.. maxlength;
```



```
               trs: ARRAY [1.. maxlength] OF treept
           END;
    newcharset: ARRAY [alphabet] OF newchar;
    stack: newchar;


PROCEDURE get_char;

    BEGIN
        REPEAT  num_in_chars := num_in_chars + 1;   get(in_file)
        UNTIL eof(in_file) OR (ord(in_file^) <> 0)
    END (GET_CHAR);


PROCEDURE fill_tally;
(Scan the file the first time and get a character count on
which to make the new Huffman character set.)

    VAR
        count: integer;

    BEGIN
        reset(in_file);
        FOR count := minchar TO 127 DO
            WITH tally[count] DO
                BEGIN  marked := false;   num_of := 0  END;
        IF NOT eof(in_file)
        THEN
            BEGIN
                get_char;
                WHILE NOT eof(in_file) DO
                    BEGIN
                        IF ord(in_file^) < minchar  THEN
                            BEGIN
                                writeln(tty, 'Bad character in input!^G', ord(
                                  in_file^));
                                GOTO 13
                            END;
                        tally[ord(in_file^)].num_of := tally[ord(in_file^)
                          ].num_of + 1;
                        get_char;
                    END
            END
    END (FILL_TALLY);


PROCEDURE exchange(VAR x, y: integer);

    VAR
        temp: integer;

    BEGIN  temp := x;   x := y;   y := temp  END (EXCHANGE);


PROCEDURE make_new_chars;
(Make the Huffman characters based on the character frequencies
of the file.)

    VAR
        temp: treept;
        posi, pos2, count: integer;
        done, tri, tr2: boolean;


    PROCEDURE ground(t: treept; val: integer);
(Ground the character tree with NIL's)

        BEGIN
            WITH t^ DO
                BEGIN  left := NIL;   right := NIL;   sum := val  END
        END (GROUND);


    PROCEDURE get_2_mins(VAR pos1, pos2: integer; VAR tr1, tr2:
      boolean);
(Find the two characters or character trees with the smallest
frequencies.)

        VAR
            count, min1, min2: integer;

        BEGIN (GET_2_MINS)
            min1 := maxint;   min2 := maxint;
            FOR count := minchar TO 127 DO
                WITH tally[count] DO
                    IF NOT marked
                    THEN
                        IF num_of < min2
                        THEN
                            IF num_of < min1
                            THEN
                                BEGIN
                                    min2 := min1;   tr2 := tr1;
                                    pos2 := pos1;   pos1 := count;
                                    tr1 := false;   min1 := num_of
                                END
                            ELSE
                                BEGIN
                                    pos2 := count;   tr2 := false;
                                    min2 := num_of
                                END;
            FOR count := 1 TO trees.t_num DO
                WITH trees.trs[count]^ DO
                    IF sum < min2
                    THEN
                        IF sum < min1
                        THEN
                            BEGIN
                                min2 := min1;   tr2 := tr1;   pos2 := pos1;
                                pos1 := count;   tr1 := true;   min1 := sum
                            END
                        ELSE
                            BEGIN
                                min2 := sum;   tr2 := true;   pos2 := count
                            END;
            IF NOT tr1   THEN  tally[pos1].marked := true;
            IF NOT tr2   THEN  tally[pos2].marked := true
        END (GET_2_MINS);
```

```
141     BEGIN (MAKE_NEW_CHARS)
142        trees. t_num := 0;
143        REPEAT
144           get_2_mins(pos1, pos2, tr1, tr2);
145           IF tr1 AND tr2
146           THEN
147              WITH trees DO
148                 BEGIN
149                    IF pos2 < pos1   THEN  exchange(pos1, pos2);
150                    new(temp);
151                    temp^. sum := trs[pos1] ^. sum + trs[pos2] ^. sum;
152                    temp^. left := trs[pos1];
153                    temp^. right := trs[pos2];   trs[pos1] := temp;
154                    t_num := t_num - 1;
155                    FOR count := pos2 TO t_num DO
156                       trs[count] := trs[count + 1]
157                 END
158           ELSE
159              IF NOT tr1 AND NOT tr2
160              THEN
161                 WITH trees DO
162                    BEGIN
163                       t_num := t_num + 1;   new(trs[t_num]);
164                       WITH trs[t_num] ^ DO
165                          BEGIN
166                             sum := tally[pos1]. num_of + tally[pos2].
167                               num_of;
168                             new(left);   new(right);
169                             ground(left, pos1);   ground(right, pos2)
170                          END
171                    END
172              ELSE
173                 WITH trees DO
174                    BEGIN
175                       IF tr2   THEN  exchange(pos1, pos2);
176                       new(temp);
177                       temp^. sum := trs[pos1] ^. sum + tally[pos2].
178                         num_of;
179                       temp^. left := trs[pos1];   new(temp^. right);
180                       ground(temp^. right, pos2);   trs[pos1] := temp
181                    END;
182           done := true;
183           FOR count := minchar TO 127 DO
184              done := done AND tally[count]. marked
185        UNTIL done AND (trees. t_num = 1)
186     END (MAKE_NEW_CHARS);
187
188
189  PROCEDURE get_new_char_set;
190  (Take the Huffman character set out of tree form and into array
191  form, so as to make accessing easier.)
192
193
194     PROCEDURE next_char(tpt: treept);
195
196        BEGIN
197           IF tpt^. right <> NIL
198           THEN
199              WITH stack DO
200                 BEGIN
241                    length := length + 1;   nchar[length] := 0;
242                    next_char(tpt^. right);   nchar[length] := 1;
243                    next_char(tpt^. left);   length := length - 1
244                 END
245           ELSE  newcharset[tpt^. sum] := stack
246        END (NEXT_CHAR);
247
248
249     BEGIN (get_new_char_set)
250        stack. length := 0;   next_char(trees. trs[1])
251     END (get_new_char_set);
252
253
254  PROCEDURE put_word(i: bit);
255  (Add a bit to the output buffer word and print when full.)
256
257     BEGIN
258        pos := pos + 1;   wd[pos] := i;
259        IF pos = bit_size  THEN
260           BEGIN
261              num_out_words := num_out_words + 1;   pos := 0;
262              out_file^ := wd;   put(out_file)
263           END
264     END (PUT_WORD);
265
266
267  PROCEDURE flush;
268  (Print out the final word, preceded by its length.)
269
270
271     PROCEDURE convert(i: integer; VAR w: out_word);
272
273        VAR
274           con: RECORD
275                 CASE boolean OF
276                    true: (j: integer) (Note: it is assumed that
277                    an integer takes up exactly one word.);
278                    false: (wd: out_word)
279                 END;
280
281        BEGIN  con. j := i;   w := con. wd  END (CONVERT);
282
283
284     BEGIN (FLUSH)
285        IF pos <> 0
286        THEN
287           BEGIN
288              num_out_words := num_out_words + 1;   out_file^ := wd;
289              put(out_file)
290           END
291        ELSE  pos := bit_size;
292        num_out_words := num_out_words + 1;   convert(pos, wd);
293        out_file^ := wd;   put(out_file)
294     END (FLUSH);
295
296
297  PROCEDURE write_integer(i: integer);
298  (Print an integer bit by bit.)
299
300     VAR
```



```
301        pow_2: integer;
302
303     BEGIN
304        pow_2 := maxdepth;
305        REPEAT
306           put_word(i DIV pow_2);   i := i - (i DIV pow_2) * pow_2;
307           pow_2 := pow_2 DIV 2
308        UNTIL pow_2 = 0
309     END (WRITE_INTEGER);
310
311
312  PROCEDURE put_new_char(VAR ch: newchar);
313  (Print a Huffman character.)
314
315     VAR
316        count: integer;
317
318     BEGIN
319        WITH ch DO
320           FOR count := 1 TO length DO
321              BEGIN  put_word(nchar[count])  END
322     END (PUT_NEW_CHAR);
323
324
325  PROCEDURE init_out;
326  (Print the generated Huffman character set into the beginning of
327  the file, so that "RECALL" can restore the file.)
328
329     VAR
330        i, j: integer;
331
332     BEGIN
333        rewrite(out_file);
334        FOR i := minchar TO 127 DO
335           BEGIN
336              write_integer(newcharset[i]. length);
337              put_new_char(newcharset[i])
338           END
339     END (INIT_OUT);
340
341
342  PROCEDURE translate;
343  (Scan the file a second time, only change from the standard
344  character set to the new one.)
345
346     BEGIN
347        init_out;   reset(in_file);
348        IF NOT eof(in_file)
349        THEN
350           BEGIN
351              IF ord(in_file^) = 0   THEN  get_char;
352              WHILE NOT eof(in_file) DO
353                 BEGIN
354                    put_new_char(newcharset[ord(in_file^)]);
355                    get_char;
356                 END;
357              flush
358           END
359     END (TRANSLATE);
360
361
362  PROCEDURE print_stats;
363  (Print the number percentage of pages saved.  Note:  The DEC-20
364  stores files by units of pages which are 512 words each.)
365
366
367     FUNCTION pages(i: integer): integer;
368
369        BEGIN
370           IF i MOD 512 = 0   THEN  pages := i DIV 512
371           ELSE  pages := i DIV 512 + 1
372        END (PAGES);
373
374
375     BEGIN (PRINT_STATS)
376        num_in_chars := num_in_chars DIV 2;
377        IF num_in_chars MOD 5 = 0
378        THEN  num_in_chars := num_in_chars DIV 5
379        ELSE  num_in_chars := num_in_chars DIV 5 + 1;
380        writeln(tty, 'There has been a ', ((pages(num_in_chars) - pages
381          (num_out_words)) / pages(num_in_chars)) * 100: 2: 1,
382          '% saving on your file.')
383     END (PRINT_STATS);
384
385
386  BEGIN (MAIN)
387     writeln(tty,
388       'Version 2.02 of Compress');
389     pos := 0;   num_in_chars := 0;   num_out_words := 0;
390     writeln(tty, 'Scanning.');   fill_telly;
391     writeln(tty, 'Calculating.');   make_new_chars;
392     get_new_char_set;   writeln(tty, 'Compressing.');   translate;
393     print_stats;   !31
394  END (MAIN).
```

```
{
    Written by:     Tom Slone
                    Nov 15, 1980
                    at Lehigh University,
                    Bethlehem, PA 18015
                    on a DEC System 20

    (c) Copyright 1980

The author grants permission to copy for non-profit use, providing
this comment remains.

}
PROGRAM recall(in_file, out_file);
{
    This program reads the Huffman codes printed in the
beginning of a file produced by the sister program, "COMPRESS"
and restores the rest of the file to its original form.
}

LABEL
    13;

CONST
    minchar = 1 {This is the minimum recognizable character
                 (nulls are ignored)};
    maxdepth = 64 {This number should correspond to the one for
                   maxdepth in "COMPRESS"};
    maxlength = maxdepth;
    maxint = 34359738367;
    bit_size = 36 {This number should correspond to the one for
                   bit_size in "COMPRESS"};

TYPE
    bit = 0.. 1;
    in_word = PACKED ARRAY [1.. bit_size] OF bit;
    alphabet = minchar .. 127;
    old_char = RECORD
                   length: 0.. maxdepth;
                   nchar: PACKED ARRAY [1.. maxdepth] OF bit
               END;
    treept = ^ tree;
    tree = RECORD
               CASE fruit: boolean OF
                   true: (ch: alphabet);
                   false: (left, right: treept)
           END;

VAR
    in_file: FILE OF in_word;
    out_file: text;
    branch: treept;
    inp1, inp2: in_word;
    num_left, pos: 0.. bit_size;
    hay_dos, done: boolean;
    depth: integer;

PROCEDURE init;

    BEGIN
        new(branch);   branch^.fruit := false;   branch^.left := NIL;
        branch^.right := NIL;   reset(in_file);   inp1 := in_file^;
        get(in_file);   inp2 := in_file^;   get(in_file);   pos := 1;
        hay_dos := true;   done := false
    END (INIT);


FUNCTION get_bit: bit;

    VAR
        con: RECORD
                 CASE boolean OF
                     true: (int: integer);
                     false: (w: in_word)
             END;

    BEGIN
        IF NOT eof(in_file)
        THEN
            IF pos < bit_size
            THEN  BEGIN  get_bit := inp1[pos];   pos := pos + 1  END
            ELSE
                BEGIN
                    get_bit := inp1[bit_size];   pos := 1;   inp1 := inp2;
                    inp2 := in_file^;   get(in_file)
                END
        ELSE
            BEGIN
                IF hay_dos  THEN
                    BEGIN
                        con.w := inp2;   num_left := con.int + pos - 1;
                        hay_dos := false
                    END;
                get_bit := inp1[pos];
                IF pos = num_left   THEN  done := true
                ELSE  pos := pos + 1
            END
    END (GET_BIT);


PROCEDURE fill_tree;

    VAR
        i: integer;
        save_tree: treept;


    FUNCTION get_integer: integer;

        VAR
            pow_2, ans, count: integer;

        BEGIN
            pow_2 := maxdepth;   ans := 0;
            FOR count := 1 TO 7 DO
                BEGIN
                    ans := ans + pow_2 * get_bit;   pow_2 := pow_2 DIV 2
                END;
            get_integer := ans
        END (GET_INTEGER);


    PROCEDURE add_one(num_left: integer; VAR kh: alphabet; VAR tr:
      treept);


        PROCEDURE start(VAR t: treept);

            BEGIN
                IF t = NIL  THEN
                    BEGIN
                        new(t);   t^.fruit := false;   t^.left := NIL;
                        t^.right := NIL
                    END
            END (START);


        BEGIN (ADD_ONE)
            depth := depth + 1;
            IF depth > maxdepth  THEN
                BEGIN
                    writeln(tty,
                        'Your file is not compatible with this program!'G');
                    GOTO 13
                END;
            IF num_left = 0
            THEN  BEGIN  tr^.fruit := true;   tr^.ch := kh  END
            ELSE
                IF get_bit = 0
                THEN
                    BEGIN
                        start(tr^.left);
                        add_one(num_left - 1, kh, tr^.left)
                    END
                ELSE
                    BEGIN
                        start(tr^.right);
                        add_one(num_left - 1, kh, tr^.right)
                    END;
            depth := depth - 1
        END (ADD_ONE);


    BEGIN (FILL_TREE)
        save_tree := branch;
        FOR i := minchar TO 127 DO  add_one(get_integer, i, branch);
        branch := save_tree
    END (FILL_TREE);


PROCEDURE translate;


    PROCEDURE convert(t: treept);

        BEGIN
            IF t^.fruit   THEN  write(out_file, chr(t^.ch))
            ELSE
                IF done
                THEN  writeln(tty, 'Warning!  Character mismatch!'G')
                ELSE
                    IF get_bit = 0   THEN  convert(t^.left)
                    ELSE  convert(t^.right)
        END (CONVERT);


    BEGIN (TRANSLATE)
        rewrite(out_file);   WHILE NOT done DO  convert(branch)
    END (TRANSLATE);


BEGIN (RECALL)
    writeln(tty,
      'Version 2 of Recall (Not compatable with version 1)!');
    writeln(tty, 'Initializing.');   init;   depth := 0;   fill_tree;
    writeln(tty, 'Recalling.');   translate;   13:
END (RECALL).
```


Articles

The Performance of Three CP/M-Based Pascal Translators

Mark Scott Johnson and Thomas O. Sidebottom
106 Mission Drive
Palo Alto, California 94303

1981 October

Abstract

The translation-time and run-time performance of three CP/M-based Pascal translators — Sorcim's Pascal/M, MT MicroSYSTEMS' Pascal/MT+, and Ithaca InterSystems' Pascal/Z — are compared. Using a benchmark of eight programs on a 4MHz Z80-based microprocessor, we find that Pascal/M excels in translation time and that Pascal/Z excels in run time. Pascal/MT+'s translation time approaches that of Pascal/M for long programs. Several translator limitations are also illustrated by the benchmark.

Introduction

We recently had the opportunity to use and evaluate four micropro essor-based Pascal translators. We are reporting here the results of one aspe-t of this evaluation (namely, performance) for three of them.

The *performance* of a piece of software, such as a programming language translator, is measured in terms of the amount of resources required by the software to produce some useful result. The primary resource we are interested in is time. We measured both the time required to translate a source program into a machine-executable form and the time required to execute the translated program. The former is termed *translation time* and the latter *run time* (or execution time).

The three Pascal translators we evaluated are Sorcim's Pascal/M, MT MicroSYSTEMS' Pascal/MT+, and Ithaca InterSystems' Pascal/Z. All three run under Digital Research's CP/M operating system. We also evaluated a fourth translator, the UCSD Pascal system, which runs under its own operating system. We have excluded UCSD Pascal from our report because we do not feel a fair comparison of translator performance can be made across operating systems. Separating the performance attributable to the operating system from that attributable to the translator is a difficult task. Other translators beside these three run under CP/M, however. We limited the study to translators that accept essentially the full Pascal programming language and that are widely accessible to the general microcomputing public.

Thus we eliminate from consideration several "tiny" Pascal translators, among others.

Translators

To better understand the behavior of the three Pascal translators and to better appreciate the performance results, we begin with a brief introduction to translator construction. We use *translator* in the generic sense — any software system that accepts as input a program in one language (the *source language*) and that produces as output a functionally equivalent program written in another language (the *object language*). If the source language is a high-level language such as Pascal and the object language is a low-level language such as assembly language or machine language, then the translator is called a *compiler*. If both the source and the object languages are low-level, then the translator is called an *assembler*. If the object language is not the machine language of some real machine, it becomes necessary to execute the object code with an *interpreter*, which simulates the object language on a real computer.

Compilers that translate source programs directly into object programs are called *one-pass* compilers. Sometimes compilers are written to perform one or more intermediate transformations between source and object; these are called *multi-pass* compilers. Multi-pass compilers generally take longer than one-pass compilers, but they often require less main memory, compile longer source programs, provide more complete diagnostics, and generate better object code. To conserve main memory (and again to increase the size of source programs that can be translated), multi-pass compilers often write out their intermediate transformations to temporary disk files.

We used version 3.19 of the Pascal/M translator. It is patterned after the UCSD Pascal system, comprising two components: a compiler that translates a Pascal source program into P-code — object code for a fictitious, Pascal-like P-machine — and an interpreter for the P-machine. It is a one-pass compiler written in Pascal. For short and moderately-sized programs the compiler uses no memory overlays, but long programs require swapping from the disk of segments of the compiler. It runs in 56K of main memory and requires no temporary files. The output from the compiler is a file containing P-code instructions, which is input to the P-machine interpreter. For compactness and efficiency, the interpreter is written in the assembly language of the host computer (a Zilog Z80, in our case).

We used version 5.2 of the Pascal/MT+ translator. It is a true compiler that generates object code for any of several microprocessors, including the Z80. It is a three-pass compiler written in Pascal: the first pass converts a source program into a sequence of logically related characters called *tokens*, the second pass builds a symbol table, and the third pass generates object code and places it in a Microsoft-format, relocatable object file. The compiler runs in 56K of main memory, using five memory overlays, and it uses one temporary file for the tokens.

We used version 3.2 of the Pascal/Z translator. It is also a compiler, but it generates an assembly-language program as its output. This assembly language requires a special assembler that is supplied with the translator, which can only generate Z80 object code. Pascal/Z is a one-pass compiler written in Pascal. It requires 56K of main memory (although 64K is recommended), using one memory overlay, and it requires no temporary files.

Benchmark

To adequately compare performance, we needed a *benchmark* — a point of reference for our measurements. A benchmark for a translator is a collection of source programs, written in the language the translator understands, that exercises various aspects of the translator's capabilities. Such benchmarks generally include short programs, long programs, and programs that stretch the limits of the translator, such as programs with deeply nested control structures or large data storage requirements. The idea is to include a mix of programs that are representative of the programs that the translator will encounter in normal, everyday use.

Rather than develop our own benchmark from scratch, we relied heavily on the work of others. In particular, seven of the eight programs in our benchmark were adapted from a performance study of the CDC 6400 Pascal translator running under the SCOPE 3.4 operating system, made several years ago by Niklaus Wirth, the designer of Pascal. We restricted our adaptations exclusively to the removal of implementation-dependent features, such as the presence of a hardware clock on the CDC 6400 and the maximum size of integers and reals. It is important to note that we made no other modifications to these programs. Several of them would not compile under one of the translators. We probably could have modified these programs to make them compilable. We optad instead to let our evaluation rest on a translator-independent benchmark.

The first benchmark is a 47-line program to compute the first 90 positive and negative powers of 2. The algorithm uses integer arithmetic exclusively, including multiplication and division. No standard Pascal functions (such as SQRT) are used, and arbitrary precision is simulated by storing each digit of the result separately in the elements of an array. "Powers of Two" is a useful benchmark since it heavily exercises integer arithmetic.

The second benchmark is a 43-line program to sort a 10,000-element array of arbitrary integers into ascending order. The sorting algorithm is called Quicksort, which relies extensively on a recursive procedure. The maximum depth of recursion is $ln(10,000)*10$. "Quicksort" is useful since it exercises recursion and array manipulation.

The third benchmark is a 32-line program to write and to read a file containing 1000 real numbers. First the numbers are written out, one per record, to a file. Then the file is reset and the numbers are read back in. The numbers are stored in internal format (that is, not in human readable form); no input/output conversions are performed. "Real IO" is useful since it exercises "naked" file handling.

The fourth benchmark is a 51-line program to solve the "eight queens" problem. The problem is to find the 92 configurations of eight queens on a chessboard such that no queen attacks another queen. The algorithm uses backtracking and recursion to exhaustively try all plausible chessboard positions. "Eight Queens" is useful since it heavily exercises iterative constructs such as for-loops and if-then-else statements, together with simple but repetitive array manipulation.

The fifth benchmark is a 47-line program to compute the first 1000 prime numbers. "Primes" uses essentially the same language features as Powers of Two, but involves more computations.

The sixth benchmark is a 29-line program to compute the ancestors of a group of individuals, given their parents. It uses a 100x100-element Boolean matrix to represent the individuals and the parent/offspring relationships among them. "Ancestor 1" is useful since it contains deeply nested control constructs and two-dimensional arrays, and thus exercises these aspects of a translator's capacity.

The seventh benchmark is a reimplementation of the previous one, using a 100-element Pascal set in place of a Boolean matrix. "Ancestor 2" is useful for comparing the performance of the implementation of sets.

The last benchmark is a 280-line program we wrote to compute the position of the moon at a given time and date. The program uses nine real arrays indexed by enumerated types, two record types, ten internal functions, and five internal procedures. Most of the functions are one-line long, and do such things as calculate the trigonometric functions in degrees and convert to and from radians and degrees. "Moon Position" is a useful benchmark since it heavily exercises real arithmetic and the compiler's capacity to handle moderately long programs.

Hardware

All of our benchmark programs were run on NorthStar Horizons, containing 4MHz Z80 microprocessors, 56K of main memory, and two double-density, single-sided 5-1/4-inch Shugart SA400 floppy disk drives. Although some of the manufacturers claim their translators will operate on smaller systems, we believe our system is the minimum configuration required for reasonable response and minimal frustration. All three translators were run under CP/M 2.2, using the NorthStar version distributed by Lifeboat Associates.

Methods

For each translator we first verified that each of the benchmark programs produced the correct results. We then removed all statements that wrote to the terminal screen, except for a WRITELN at the beginning of each program that wrote "GO" and a WRITELN at the end of each that wrote "STOP". We did not use these output messages for our measurements; they were merely to give us feedback that something was happening. To guarantee comparable run-time statistics, we compiled each program with *all* error checking, such as range checking and IO failure detection, disabled.

Because NorthStar Horizons are not equipped with hardware clocks, all timing measurements were made using a stopwatch. We timed each separate step (compile, assemble, link, and run) by typing the appropriate CP/M command line, waiting for the disk drives to stop spinning, and then simultaneously hitting the RETURN key and the start button on the stopwatch. We stopped the watch when the next CP/M command prompt ("A>") appeared. Thus all of our measurements include the time required by CP/M to process the command line, to locate and load the appropriate software into memory, and to prompt for the next command. This method does not measure the "bare bones" performance of the three Pascal translators and the object code that they produce. Nevertheless, we believe that it reflects the typical user's interactions, and thus the method accurately measures the performance that such users can expect for themselves.

Several of the measurements were taken twice to check for timing variance. In no case did the times differ by more than 0.3 seconds, which we attributed to variations in controlling the stopwatch. Thus the variance appeared insignificant.

Results

Tables 1 thru 3 show the results of translating and executing the benchmark programs with each of the three translators. Each column in the tables represents one CP/M command. Tables 4 and 5 summarize the results of the first three tables. In Table 4 translation time is computed as the sum of all the steps necessary to make the object programs executable.

Table 1 shows that Real IO would not compile under Pascal/M. Pascal/M does not support the READ and WRITE procedures on the type FILE OF REAL. As expected with an interpreter-based system, Pascal/M compiles quickly, but interpretation of the P-code is slow. Compile time remained approximately 80 lines of source code per minute, even with long programs such as Moon Position.

Pascal/MT+ successfully compiled all the benchmark programs (Table 2). Compilations are typically up to three times longer than with Pascal/M; total translation time is up to four times longer. Nevertheless, run time ranges from about 30% to 200% faster. Compile time was approximately 30 lines of code per minute for the short programs, but rose to 70 lines per minute for the long program. Total translation time was about 25 lines per minute for the short programs and 56 lines per minute for Moon Position.

Two of the programs would not compile under Pascal/Z (Table 3). Both had control structures too deeply nested (about eight levels) for the compiler to handle. Pascal/Z's compile time is only about one-third longer than Pascal/M's and about twice as fast as Pascal/MT+'s for short programs (approximately 65 lines per minute). But the extra assembly step required takes up to twice as long as the compile time. Table 4 shows that the overall translation time of Pascal/Z is three to four times slower than Pascal/M and ranges from about 25% to 200% slower than Pascal/MT+. Translation time for long programs decreased slightly (25 lines per minute as opposed to 20 lines per minute). Nevertheless, Pascal/Z consistently produced faster code than did Pascal/MT+, ranging from about 10% to 150% faster.

Conclusions

For applications that require frequent compilation but infrequent execution, or where run-time speed is unimportant, Pascal/M is a good choice.

Pascal/Z is the best alternative when run-time performance is paramount and your code only needs to run on Z80s. But be prepared for excruciatingly slow translation time, especially on long programs. Also be prepared to restructure your programs to get them to compile, especially if your system has less than 64K of main memory.

Pascal/MT+ lies somewhere between these extremes. Translation time is slow, but the relative speed (that is, lines of code per minute) improves significantly as program size increases. Similarly, run time is much better than Pascal/M, but not as good as Pascal/Z for most programs. Run-time performance for the two recursive benchmarks, Quicksort and Eight Queens, was relatively poorer than for the nonrecursive benchmarks.

We conclude with a strong admonition. We have reported here only one aspect of comparison between the three translators, namely time performance. There are many other aspects that must be considered when deciding on a translator to suit your own needs, such as robustness, documentation, support, language extensions, error handling, size of object code, and ease of use. For example, in applications where reentrant code is important, Pascal/Z is the only alternative of the three. We decided on Pascal/MT+ for our own applications, primarily because of the language extensions it provides (it is the most complete systems implementation language of the three) and its robustness (we seldom have to massage our code to get it to compile).

Acknowledgements

We extend our thanks to Dionex Corporation, Hewlett-Packard Laboratories, and Pluto Research Group for access to their computers and other resources during this study.

Program	Compile Time	Run Time
Powers of Two	34.6	29.5
Quicksort	32.3	5:23.0
Real IO	unsuccessful	N/A
Eight Queens	36.1	5:02.8
Primes	33.9	1:13.8
Ancestor 1	32.3	1:51.3
Ancestor 2	31.5	43.4
Moon Position	3:30.3	17.4

Table 1: Pascal/M Timing Results (in minutes and seconds).

Program	Compile Time	Link Time	Run Time
Powers of Two	1:31.3	30.4	9.6
Quicksort	1:30.7	39.0	2:47.6
Real IO	1:26.0	38.7	37.0
Eight Queens	1:32.8	30.9	2:30.5
Primes	1:31.6	30.3	11.6
Ancestor 1	1:30.5	33.6	24.9
Ancestor 2	1:28.3	31.5	23.8
Moon Position	3:30.5	53.2	12.8

Table 2: Pascal/MT+ Timing Results (in minutes and seconds).

Program	Compile Time	Assembly Time	Link Time	Run Time
Powers of Two	44.8	58.0	46.8	8.9
Quicksort	43.3	59.3	48.6	1:05.5
Real IO	38.3	58.9	56.1	20.9
Eight Queens	48.0	1:04.0	49.7	53.4
Primes	unsuccessful	N/A	N/A	N/A
Ancestor 1	unsuccessful	N/A	N/A	N/A
Ancestor 2	41.8	1:03.2	46.3	18.0
Moon Position	3:34.6	6:06.5	1:42.1	10.5

Table 3: Pascal/Z Timing Results (in minutes and seconds).

Program	Lines	Pascal/M	Pascal/MT+	Pascal/Z
Powers of Two	47	34.6	2:01.7	2:29.6
Quicksort	43	32.3	2:09.7	2:31.2
Real IO	32	N/A	2:04.7	2:33.3
Eight Queens	51	36.1	2:03.7	2:41.7
Primes	47	33.9	2:01.9	N/A
Ancestor 1	29	32.3	2:04.1	N/A
Ancestor 2	29	31.5	1:59.8	2:31.3
Moon Position	280	3:30.3	4:52.7	1:23.2

Table 4: Summary of Translation-Time Results (in minutes and seconds).

Program	Pascal/M	Pascal/MT+	Pascal/Z
Powers of Two	29.5	9.6	8.9
Quicksort	5:23.0	2:47.6	1:05.5
Real IO	N/A	37.0	20.9
Eight Queens	5:02.8	2:30.5	53.4
Primes	1:13.8	11.6	N/A
Ancestor 1	1:51.3	24.9	N/A
Ancestor 2	43.4	23.8	19.0
Moon Position	17.4	12.8	10.5

Table 5: Summary of Run-Time Results (in minutes and seconds).

****

MACALESTER COLLEGE
1600 GRAND AVENUE
SAINT PAUL, MINNESOTA 55105
612 696-6000

October 7, 1981

Mr. Rick Shaw
Pascal Users Group
P. O. Box 888524
Atlanta, Georgia 30338

Dear Mr. Shaw:

The enclosed article reports my reactions and those of my students to the first Pascal programming course that I taught. I am fairly new to the field of computer science and this particular teaching experience was exciting to say the least.

I hope this short piece will prove to be of interest to you and your readers.

Sincerely,

Jerry
Gerald R. Pitzl
Associate Professor

GRP:ba

Encl.

MACALESTER COLLEGE

A Geographer Teaches Pascal -- Reflections on the Experience

Jerry Pitzl
Macalester College
St. Paul, Minnesota

Macalester College, a small (1700 students), liberal arts institution located in St. Paul, Minnesota, recently initiated a new major in Computer Studies. Several courses in programming have been offered over the years but increased student demand for a wider range of offerings and faculty recognition that a full and complete program would be necessary in order for us to keep pace with the rapidly growing field of computer science necessitated this significant change.

As a further enhancement to the computer program, Macalester College, in 1979, became the recipient of a National Science Foundation grant to be used to expand the use of computers within science laboratory settings. Initial purchases of hardware included three DEC MINC-11 computers especially configured for laboratory applications. In addition, the departments of geography, of which I am a member, and geology received a Magnavox 5-4 Orion stand-alone graphics system, a 22" x 22" Talos S622 digitizer, and a 300 LPM Printronix Printer/Plotter. The graphics system is used primarily within the geography department in a computer mapping course.

During the academic year 1979-80 I was on a sabbatical leave and spent virtually all my time at the University of Minnesota auditing courses in a variety of computer and mathematics related areas. I had no prior knowledge of computer languages, but I knew that I would have to become familiarized as quickly as possible because I was slated to do the computer mapping course. Needless to say, the transition to the "kind of thinking" required for success in the computer field did not come that easily for me at first; my long-term background, primarily in the humanistic realms of geography, had produced a "mind set" that was placed in a mild form of intellectual shock at first exposure to computer operations, and this condition persisted for at least the first few weeks.

Fortunately, however, my introduction to computer programming was through the Pascal language. I found the language to be logically constructed and relatively easy to use. The form of program development using algorithm formulation and structure provided an ideal transition to the eventual writing of actual Pascal code. I soon became unequivocally "hooked" on Pascal. So much so that in the following year I set out to develop a course in programming with Pascal which was introduced during our January "interim" session of 1981. Interim is a one-month period in which courses not available in the regular semesters are given. It is a good time to introduce and test a topic or theme which may later become a regular curricular offering. In our case, Pascal was not a new topic on campus; it is being taught along with other languages in a one-semester course. However, I felt that the language should receive a great deal more emphasis and perhaps eventually be the sole subject of a full semester. It is, as most agree, the most appropriate language for teaching the concepts of structured programming.

The interim course contained 20 students, half of whom had varying degrees of experience with computer science and the rest with no experience whatever. The four-week time frame with two-hour sessions five days a week left little free time for either the students or the instructor. We covered all aspects of the language including a brief introduction to the use of records, external files and the pointer.

The students produced eight programs of varying difficulty and took four quizzes. The assigned readings came from Schneider, Weingart, and Perlman, Introduction to Programming and Problem Solving with Pascal, a widely used and thorough introduction to the language. As an added feature, G. Michael Schneider, one of the authors of the text, visited the class and gave us a most stimulating presentation.

As a final exercise in the course, the students were asked to complete a critique of the experience. Some of the questions asked and a sampling of the responses are presented here:

Item No. 1--Did you know a programming language before this course?

--- There were no "no's" ---

a. If yes, how would you compare Pascal to the language(s) you already know? Responses: requires new ways of thinking...about flow of control; most flexible language I know; much prettier...easy to use and efficient once the bad habits of needing the "go to" statement are broken; easier to understand than COBOL or FORTRAN; more high-powered than BASIC and more structured; more can be done with Pascal; more ways to approach a problem; compared to BASIC, Pascal is much more fun; more closely related to the English language.

b. If no, did you find that Pascal provided a meaningful introduction to programming? If yes, why? If no, why not? Responses: Yes, I think the structure is important; yes, it provides the basis for a new way of thinking; yes, good intro to the computer and how it works; yes, judging from the experiences of those in the terminal room using other languages, it seems that Pascal is the best language for understanding programming; yes, it is easy to work with; yes, Pascal has provided me with a meaningful introduction to programming; yes, it is easy to read a program...and the language is interesting; yes, Pascal was a good introduction in that I learned that programming is mostly paperwork before hand.

Item No. 7--Do you think that Pascal should be offered as a full, regular semester course? If yes, please state why; if no, please state why not.

Responses: Yes-- interesting, powerful; important for computer studies majors; good for structured programming; it is a relatively new language and computer studies majors should know it; it is the direction that computer languages will go; best for general purpose computing; becoming more widely accepted and used; valuable course for learning many aspects of computer science; better for beginners -- neat, beautiful language; more time needed than is available during interim; versatility and uses of the language are great; better to learn as a "first" language; a "fun" language; a "logical" language; very powerful.

Item No. 9--Do you think that you will choose to use Pascal in the future if you write computer programs?

--- All yes's ---

How would you rate our guest lecturer, Professor Schneider?

Responses: good, excellent; interesting; informative; amusing; a good prospect for a Mac prof; excellent; very knowledgeable; knows his stuff; great future; very good; great -- too bad we can't be assured of having him here; great teacher; 8 on a scale of 10; excellent; he really knows his stuff; excellent; 10 of 10; great; great guy; really knows what he's doing; liked him; slick and intelligent guy; fantastic; sparked my interest in computer science; the high point of the class; he is like the pointer -- dynamic.

Final Item--General comments.

Responses: best interim course ever taken; more challenging than BASIC; impressive language; I now have an understanding and a respect for computers; revived my ability to concentrate for extended periods of time; computers -- "it's rather amazing, isn't it?"

As the responses clearly suggest, the entire class was more than satisfied with the course and unanimous in their assessment of Pascal as a sound and usable programming language. It would be sheer understatement on my part to say that I was pleased with the outcome. I was ecstatic! The course is scheduled for the interim term of 1982 and the Pascal language offering during the regular semester will be expanded within the existing course framework.

I conclude with a plea to all who are in an academic setting to encourage the expanded offering of Pascal as the most appropriate language to use for introducing programming. I believe this to be true not only for students, but for others (faculty and staff) who are being tasked to climb aboard the expanding computer applications wave that apparently is nowhere near cresting.

****

money management systems, inc.

309 wyman street • waltham, massachusetts • 02154
(617) 890-8070

An Extension that Solves Four Problems

by Jonathan A. Yavner

1. The Dynamic Array.

The specification of dynamic arrays is currently a point of heated discussion among Pascal theorists. Pascal News #19 (labeled "17") contains eleven double-density pages of debate on the merits of the proposal contained in the DP 7185.1 standard. The most telling argument against the Sale syntax is the assertion that it is not intuitively obvious and therefore does not belong in a language whose users consider it the guardian of rational programming. The point is substantiated by the sheer prolixity of the bombast on the subject that has been published in PN, shouted across standards-committee conference tables, or otherwise made public. If the dynamic array really belongs in Pascal--and is not present because certain vociferous fanatics chanting "Stamp out the FORTRAN dinosaur!" want to make Pascal able to do everything FORTRAN can and don't care if Pascal becomes FORTRAN in the process--there has to be a better way.

2. Memory-resident Format-Conversion.

I wonder about those fanatics, though. My company produces financial database-management systems, for which one would think Pascal an ideal language, given its data-security emphasis. However, such programming requires certain features commonly available in FORTRAN and BASIC which are difficult to simulate in Pascal. Such a feature is memory-resident format conversion. In most high-level languages, format conversion is performed as an integral part of I/O. Sometimes it is necessary to perform such conversion in memory, perhaps to add commas before output or to delete them after input. For these occasions FORTRAN provides its ENCODE and DECODE statements. BASIC implementations tend to have two or more string functions (with different names and formats for each implementation) to perform these conversions. I hear no fanatic-talk about adding these features to Pascal, yet the only way to force Pascal to perform non-I/O conversion is to declare an external procedure and then attach it to the appropriate routine in the run-time-library using some sort of aliasing mechanism--an extremely implementation-dependent method. If the implementation doesn't support external procedures or doesn't list the names of its library routines or doesn't allow them to be called by the user, the program must contain a source-code duplicate of the conversion routine--an extremely inefficient method.

This conversion problem is actually a special case of a more basic difficulty which has received occasional mention in this journal (though I can't find the references). Programming generality can be promoted by avoiding an either/or choice for main versus peripheral memory storage of files. In one of the references which I can't find, IBM's 48-bit unified addressing scheme is given as an example of where the capability to code storage-location-independent routines is provided to the assembly programmer.

3. The String.

Anyone who uses a version of BASIC (among others) that has a garbage collector becomes addicted to strings and finds Pascal and FORTRAN irritatingly restrictive. Like its close relative the dynamic array, there seems to be no obvious method of specifying string definition and manipulation.

4. Random-access I/O.

Pascal can be implemented on any computer with at least a processor and two magtapes. Such a computer is incapable of random-access I/O. For this reason no mention of such I/O appears in the standard. For this reason each of the vast majority of implementations which can supply random access has implemented incompatible extensions to provide this capability. The standard would be superior if there were some way to specify the format of such operations without either requiring them of all implementations or layering the standard. Use of a layered standard to define a language which includes intuitive obviousness among its design goals is a paradox.

5. The Solution.

The solution to the problems delineated above lies in the realization that dynamic arrays, strings, and files are but different facets of the same data structure. Simply extending slightly the definition of the file structure would allow files to perform the duties of strings and dynamic arrays. To avoid actually implementing garbage collection, files could be allocated in segments on the heap, each segment containing x sequences of the file and a pointer to the next segment, where x is determined from the equation

x=⌊(nice segment size)-(pointer size)⌋ DIV (sequence size).

Deletions from the standard: All references to comformant arrays, comformant array schemata, and compliance levels.

Changes to the standard, 6.4.3.5 (file types): The file element f.M has the enumerated values (Generation, Inspection, Direct). There exists an element f.Len whose value is the number of sequences in the file. The notation f[n] denotes the nth sequence of the file; the values for n are 0..(f.Len-1). There exists an element f.Pos,

whose value is such that f.R.first=f[f.Pos]. F.Pos shall be equal to f.Len if f.R=S(). Rule (b), describing the structure of a file of type text in Generation mode, shall apply also for Direct mode.

Changes to the standard, 6.6.5.2 (file-handling procedures):

```
get(f): If f0.M=Direct,
  pre-assertions:
    f0.L is defined
    f0.R<>S()
  post-assertions:
    f.M=f0.M
    f.Len=f0.Len
    f.Pos=f0.Pos+1
    f.L=f0.L^f0.R.first
    f.R=f0.R.rest
    If f.R<>S()
      f↑=f.R.first
    otherwise
      f↑ is undefined

put(f): If f0.M=Direct,
  pre-assertion:
    f0.R, f0.L, and f0↑ are defined
  post-assertions:
    f.M=f0.M
    f.Pos=f0.Pos+1
    f.L=f0.L^S(f0↑)
    f.R=f0.R.rest
    If f0.R=S(),
      f.Len=f0.Len+1
    otherwise
      f.Len=f0.Len
    If f.R<>S()
      f↑=f.R.first
    otherwise
      f↑ is undefined
```

Additions to the standard, 6.6.5.2:

```
init(f)
  pre-assertion:
    true
  post-assertions:
    f.M=Direct
    f.L=f.R=S()
    f↑ is undefined
    f.Len=f.Pos=0

seek(f,p)
  pre-assertions:
    f0.L and f0.R are defined
    f0.M IN [Direct]+seekmodes
    p IN [0..f0.Len]
```



```
post-assertions:
  f.M=f0.M
  f.Len=f0.Len
  f.Pos=p
  f.L=f[0]^f[1]^f[2]^...^f[p-1]   (f.L=S() if p=0)
  f.R=f[p]^f[p+1]^...^f[f.Len-1]  (f.R=S() if p=f.Len)
  if f.R<>S()
    f↑=f.R.first
  otherwise
    f↑ is undefined
```

The implementation-defined set seekmodes shall be equivalent to the set of values for f.M other than Direct for which seek shall be valid.

The procedures d<fctn>(f,P,v1,v2,...,vn), where <fctn> shall be replacable by any of (read, write, readln, writeln), shall be equivalent to

```
begin seek(f,p); <fctn>(f,v1,v2,...,vn) end.
```

Additions to the standard, 6.6.5.4 (ordinal functions):

length(f) The function shall return the value of the element f.Len of file f; the set of values for f.M other than Direct for which f.Len is defined shall be implementation-defined.

pos(f) The function shall return the value of the element f.Pos of file f; the set of values for f.M other than Direct for which f.Pos is defined shall be implementation-defined.

6. Example Program.

This program fragment uses many facets of the extension outlined above. It has not been parsed, since currently there is no processor which accepts the extension. It is asserted that one of Pascal's greatest strengths lies in its ability to make this kind of general-purpose program reasonably portable. Comments would be appreciated, as it is conceivable that I may inflict upon the world a Pascal processor with this extension unless either I am drowned in a sea of hate mail or the proposal ceases to be an extension.

```
program MoneyMarketIII(input,output);

const
  ScreenHeight = 24;
  ScreenWidth  = 79;
  MaxField     = 32;
  MaxScale     = 9;
type
  Whole        = 0..MaxInt;
  Short        = -32768..32767;
```

```
  Byte        = 0..255;
  SHIndex     = 1..ScreenHeight;
  SWIndex     = 1..ScreenWidth;
  ScaleIndex  = -MaxScale..MaxScale;
  FieldTypes  = ( A,B1,B2,B4,D,X );
  TableTypes  = ( Control,FieldDesc );
  Date        = packed record
    year      : 1901..2100;
    month     : 1..12;
    day       : 1..31;
    end;
  TypeCross   = packed record { All implementation-dependent trickery
    goes through this type, thus isolating the programming changes
    necessary to move to a new processor. }
    case FieldTypes of
    A  :( aval  : Real );
    B1 :( b1val : Byte );
    B2 :( b2val : Short );
    B4 :( b4val : Integer );
    D  :( dval  : Date );
    X  :( xval  : packed array[1..MaxField] of Byte );
    end;
  TableRec    = packed record
    case rectype:TableTypes of
    Control   :( { Control record for each data file }
      name : packed array[1..8] of Char;
      fd   : Whole { Pointer to first field descriptor };
      nent : Short { Number of field descriptor entries };
      );
    FieldDesc :( { Descriptor for each field in data record }
      fx     : Short { Field number };
      ft     : FieldTypes;
      af     : Short { Auxillary field-type datum };
      loc    : Short { Location of field };
      leng   : Byte  { Length   of field };
      p      : Byte  { Screen page of fields };
      vx,vy  : Byte  { Co-ordinates of value field };
      nx,ny  : Byte  { Co-ordinates of name  field };
      name   : packed array[1..12] of Char;
      );
    end;
  Datafile = packed file of Byte;
  TableFile  = file of TableRec;
var
  filcon : TableRec { File control record };
  table  : TableFile;
  data   : DataFile;
  filnum : Byte { Data-file number };
  page   : Byte;
  ln10   : Real;

procedure Format( { ENCODE example; also shows string usage }
  var output : Text;
  input      : Real;
```



```
  scale      : ScaleIndex;
  leng       : Byte { This semicolon is illegal! --> };
  );

  var
    temp     : Text;
    nonfrac,x : Whole;
    abscale  : 0..MaxScale;
    comma    : 0..2;
  begin
    abscale:=abs(scale) { Number of implied fractional digits };
    init(output);
    write(output,exp(ln(abs(input))-abscale*ln10)::1:abscale);
    nonfrac:=length(output)-abscale-1 { Non-fractional digits };
    comma:=(nonfrac-1) MOD 3;
    init(temp);
    seek(output,0);
    for x:=1 to nonfrac do begin
      write(temp,output↑);
      get(output);
      if comma>0 then comma:=comma-1 else begin
        if x<>nonfrac then write(temp,',');
        comma:=2;
        end;
      end;
    if scale<0 then begin { Truncate decimal }
      x:=length(output);
      repeat
        x:=x-1;
        seek(output,x);
        until output↑<>'0';
      if output↑<>'.' then begin
        seek(output,nonfrac);
        for x:=nonfrac to x do begin
          write(temp,output↑);
          get(output);
          end;
        end;
      end
    else if scale>0 then while NOT eof(output) do begin
      write(temp,output↑);
      get(output);
      end;
    init(output) { Space should be recovered here };
    x:=leng-length(temp)-ord(input<0);
    if x>0 then write(output,' ':x);
    if input<0 then write(output,'-');
    while NOT eof(temp) do begin
      write(output,temp↑);
      get(temp);
      end;
    if length(output)>leng then begin
      init(output);
      for x:=1 to leng do write(output,'*');
```



```
    end;
  { System must dispose of local files here. }
  end;

procedure FormatDate(var output:Text; input:Date);
  begin
    init(output);
    with input do write(output,day:2,'/',month:2,'/',year:4);
    seek(output,3);
    if output↑=' ' then write(output,'0');
    end;

procedure Dump(var output,input:Text);
  { Generalized procedure to trim trailing blanks.  This routine is
    completely device-independent.  Output is assumed to be open. }
  label 1;
  var
    temp : Text;
    x    : Whole;
  begin
    reset(input) { Reset must perform a writeln if necessary };
    page(output) { Must also writeln };
    while NOT eof(input) do begin
      init(temp);
      while NOT eoln(input) do begin { Note the use of the end-of-line
        character as a flag.  Similar use of the end-of-page character
        is impossible because of the lack of the eop() function. }
        write(temp,input↑);
        get(input);
        end;
      readln(input);
      if length(temp)=0 then goto 1;
      repeat seek(temp,pos(temp)-1) until temp↑<>' ' OR pos(temp)=0;
      if temp↑=' ' then goto 1; x:=pos(temp); seek(temp);
      for x:=1 to x do begin
        write(output,temp↑);
        get(temp);
        end;

  1:
      writeln(output);
      end;
    end;

  procedure FillScreen( { Format and print a record }
    var output    : Text;
    var table     : TableFile { Possibly peripheral; so what? };
    var data      : DataFile { Almost certainly peripheral; requires
      that seek() be allowed on files which are associated with an
      external storage device and are in Inspection mode. }
    var tablentry : TableRec;
    page          : Byte;
    );
```



```
  var
    screen,field : Text;
    i,j,base     : Whole;
    convert      : TypeCross;

  procedure Posit(var output,input:Text; x:SWIndex; y:SHIndex);
    { Dynamic array example.  Input's maximum size depends on whether
      it is a value or a name.  Note that, in contrast to the con-
      formant array, a file argument can be packed (Text = packed
      file of char), but it cannot be passed by value, since allowing
      files to be assignment-compatible would create an ambiguity
      either of whose resolutions contains a paradox.  Oh well, such
      are the breaks . . . }
    begin
      seek(output,y*80+x) { Note that an end-of-line, in conformance
        to the standard, is assumed to occupy one sequence in the
        file.  Some ASCII computers use the old-fashioned chr(13)-
        chr(10) terminator instead of the ANSI-standard chr(10).
        Some computers have weird character sets that require escapes
        to enable certain subsets.  Many EBCDIC computers derive eoln
        from (file-position MOD record-length).  Such difficulties
        may force some implementations to prohibit the use of seek()
        on externally-associated textfiles and to use special-case
        Direct-mode-only code in all the file-handling procedures
        to produce extra-wide characters with special bits to indicate
        prefixes.  Ugh.  As I have suggested, my extension simplifies
        the programmer's job at the expense of creating double the work-
        load for the run-time library.  But anyone afraid of a little
        inefficiency should use an assembler--or a better computer!};
      reset(input);
      while NOT eof(input) do begin
        write(output,input↑);
        get(input);
        end;
      end;

  begin { FillScreen }
    init(screen);
    for i:=1 to ScreenHeight do writeln(screen,' ':ScreenWidth);
    base:=pos(data) { Assume data file already positioned };
    with tablentry, convert do begin
      seek(table,fd);
      for i:=1 to nent do begin
        with table↑ do if p=page then begin
          seek(data,base+loc);
          for j:=1 to leng do read(data,xval[j]);
          for j:=leng+1 to MaxField do xval[j]:=' ';
          case ft of
            A  : Format(field,aval,af,vl);
            B1 : Format(field,b1val,0,vl);
            B2 : Format(field,b2val,0,vl);
            B4 : Format(field,b4val,0,vl);
            D  : FormatDate(field,dval);
```

```
            X   : write(field,xval:vl);
            end;
          Posit(screen,field,vx,vy);
          init(field);
          write(field,name:nl,' (',fx:l,')');
          Posit(screen,field,nx,ny);
          end;
        get(table);
        end;
      end;
    Dump(output,screen);
    end;

begin { MoneyMarketIII }
  lnl0:=ln(10.0);
  { Determine filnum }
  dread(table,filnum,filcon);
  connect(data,filcon.name) { external is standard, why not connect? };
  reset(data) { Requires random-I/O ability in run-time environment };
  { Position datafile and determine page }
  FillScreen(output,table,data,filnum,Page,ScreenHeight,ScreenWidth);
  { Other processing }
  end.
```

7. Optional String Functions.

The main point of this essay (whenever it pretended to have one) has been that Pascal has always had string-handling ability and that the addition of a few functions could provide enough improvement to obviate any need for a heavyweight boxing match to decide which dynamic array description method should be used. However, the example program is in many ways redundant, since the same kinds of code sequences appear repeatedly. For this reason the following suggested list of string functions is proposed. Implementing them in assembly would remove the restriction that the files must be of a specific type. The "type" File, as used below, reflects this generic capability, available only to intrinsic procedures.

```
procedure Append(var output,input:File);
  begin
    reset(input);
    while NOT eof(input) do begin
      write(output,input↑);
      get(input);
      end;
    end;

procedure Copy(var output,input:File);
  begin
    init(output);
    Append(output,input);
    end;
```



```
procedure Posit(var output,input:File; sequence:Integer);
  begin
    seek(output,sequence);
    Append(output,input);
    end;

procedure Switch(var output,input:File);
  begin
    Copy(output,input);
    init(input) { Actually, since the internal pointers are being
      switched, the input file would be left undefined (closed). }
    end;

procedure Extract(var input,output:File; loc,leng:Integer);
  var x:Integer;
  begin
    seek(input,loc);
    init(output);
    for x:=1 to leng do begin
      write(output,input↑);
      get(input);
      end;
    end;

procedure Insert(var output,input:File; sequence:Integer);
  var
    temp : File;
    x    : Integer;
  begin
    Extract(output,temp,0,sequence);
    Append(temp,input);
    while NOT eof(output) do begin
      write(temp,output↑);
      get(output);
      end;
    Copy(output,temp);
    end;

function Compare(var left,right:File):1..3;
  label 1;
  begin
    reset(left);
    reset(right);
1:
    if eof(left) then Compare:=3-ord(eof(right))
    else if eof(right) then Compare:=1+ord(eof(left))
    else if left↑<>right↑ then Compare:=1+2*ord(left↑<right↑)
    else begin
      get(left);
      get(right);
      goto 1;
      end;
    end;
```



```
Function Locate(var parent,search:File):Integer;
  { Pre-assertions:  parent.M=Direct; parent.Pos is starting point. }
  { Post-assertions: parent.Pos=Locate+length(search) }
  { Locate is assigned the parent sequence number of the first element
    of search (starting the search from the input value of
    parent.Pos).  If the search file cannot be found in parent, Locate
    is returned as length(parent).  This definition avoids special-
    case handling both within Locate and in the calling code.  Compare
    this simplicity to the definition and use of DEC's BASIC instr/pos
    function! }
  label 1,2;
  var localroot:Integer;
  begin
    localroot:=pos(parent);
    while localroot<length(parent) do begin
      reset(search);
1:
      if eof(search) then goto 2;
      if eof(parent) then begin
        localroot:=length(parent);
        goto 2;
        end;
      if parent↑=search↑ then begin
        get(parent);
        get(search);
        goto 1;
        end;
      localroot:=localroot+1;
      seek(parent,localroot);
      end;
2:
    Locate:=localroot;
    end;
```

One final question: Should the first file argument of these string procedures be optional, as it is for the other intrinsic file procedures? Personally, I believe that the original file-omission option was a mistake, so I never use it. Allowing first-argument omission for the string-handling procedures would be difficult, since the second argument is often also a file. For these reasons, I vote "no."

JANER Sept 81

Open Forum For Members

BRITISH COLUMBIA HYDRO AND POWER AUTHORITY

970 BURRARD STREET
VANCOUVER, B.C.
V6Z 1Y3
TELEX 04-84386

1981 July 21

Dear PUG

subject: PRETTYPRINT

Prettyprint programs should reformat multiline comments into single line comments. This will help detect unmatched comment delimiters. It will also make it clear when bits of Pascal code are actually comments on how to modify the program.

OLD:

```
    (* TO SUM THE INTEGERS
       for i:= 1 to 10 do *)
       s:= a+w[i];
```

NEW:

```
    (* TO SUM THE INTEGERS *)
    (* for i:= 1 to 10 do *)
       s:= a+w[i];
```

"A Comment on Comments"
W. Cox
GenRad/Futuredata
17361 Armstrong Ave.
Irvine, CA 92714

Introduction

While working on our Pascal compiler for the Intel 8086 (written in UCSD Pascal), I have studied closely several of the User's Group software tools with an eye toward converting them to that dialect. I have the following observations to make regarding the handling of comments by those tools and upon the definition of a comment in the Draft Standard proposal [1].

ISO Standard Comment Forms

This table enumerates the four forms of comment permitted by the Draft Standard.

Forms	Starting Delimiter	Ending Delimiter
1	"{"	"}"
2	"(*"	"*)"
3	"{"	"*)"
4	"(*"	"}"

Note: Forms 3 & 4 are prohibited by our UCSD compiler.

UCSD Pascal Comment Handling

The UCSD Pascal compiler that we use (a much-modified version 1.5) permits Forms 1 & 2 of comments, with a most useful twist: a comment begun by a curly bracket can only be terminated by a curley bracket, and one begun by the "(*" digraph can only be terminated by the "*)" digraph. Users whose systems don't permit both forms are unaffected, but those of us who have curley bracket characters are lucky. By using only form 1 for normal comments, we are able to "comment out" our temporarily delete bodies of text (using form 2) in a natural and error-free manner.

Draft Standard Suggestion

Since the above manner of comment handling is most useful to some of us, relatively cheap to implement for all of us, and invisible to those whose character sets don't permit it, I suggest that the Draft Standard, section 6.1.8 paragraphy 1, sentence 1 be rewritten as follows:

The constructs "{"..."}" and "(*"..."*)" shall be comments if the "{" or "(*" does not occur within a character-string. The constructs "{"..."*)" and "(*"..."}" are expressly forbidden.

The note in section 6.11 should be deleted.

Software Tools Commentary

It is interesting that the software tools published in Pascal News are not uniform in their handling of comments. XREF [4], written by Pascal's inventor, and ID2ID [2] follow the UCSD convention while PRETTYPRINT [6] and REFERENCER [3] follow the Draft Standard. FORMATTER [7] doesn't recognize curley brackets at all!

References:

1. A.Addyman, et al. ISO DP/7185 -- A Draft Proposed Standard for the Programming Language Pascal.
Pascal News # 18 (May, 1980)

2. Andy Mickel. Recoding a Pascal Program using ID2ID.
Pascal News # 15 (September, 1979)

3. Sale, A.H.J. User Manual - Referencer.
Pascal News # 17 (March, 1980)

4. Wirth, N., et al. Cross Referencer Generator for Pascal Programs.
Pascal News # 17 (March, 1980)

5. Shillington & Ackland (ed). UCSD (Mini-Micro Computer).
Pascal Version 1.5 (January, 1980)
Note: This reference does not discuss the UCSD comment handling; it is included for completeness only.

6. Heuras & Ledgard. Pascal Prettyprinting Program.
Pascal News # 13 (December, 1978)

7. Condict, Marcus & Mickel. Pascal Program Formatter.
Pascal News # 13 (December, 1978)

255 Huron Street, room 360
Toronto, Ontario M5S 1A1

university of toronto
computing services

Your reference number

(416) 978-4452

Page 2

Newsaway Associates,
101 Tremont Street,
Boston MA 02108
U.S.A.

As a longtime user of the language Pascal I was interested to see a description of your language HA-PASCAL/I. As I read the description, however, I became concerned, and finally skeptical. While you have produced what will clearly be a good product and a very useful tool for the intended applications, I am concerned that you are selling a product as a Pascal language that is really not Pascal. (Pascal is not an acronym but a person's name, so it is usually written in normal case, like Ford Motors, Washington or San Diego.)

It seems to me - after reading just your advertising flyer - that your product HA-PASCAL/I is more accurately described as PL/I with some of the syntax of Pascal. Your memory references MEM and MEMW in particular use the concept of a "pseudo-variable" which is normal to PL/I but completely alien to Pascal. All Pascals that I know of use (built-in or library) FUNCTIONs and PROCEDUREs for this purpose - a FUNCTION to return a value; a PROCEDURE to send one. This is the spirit of Pascal; "pseudo-variables" are not. Also, your CALL statement for external routines is PL/I, not Pascal. All Pascals that I have used or seen declare external routines as a FUNCTION or PROCEDURE, as appropriate, with the subprogram body replaced by the keyword EXTERNAL (or EXTERN in a few cases).

Perhaps more importantly, your advertising makes no reference to the existence of the RECORD construct. The RECORD concept is one of the key concepts of Pascal; one of the things that makes Pascal Pascal. A Pascal without RECORDs is like a computer without a CPU, like a car without wheels.

Finally, your statement that the inaccessability to machine language in some Pascals (most provide it either inline or via EXTERNAL routines) prevents "the Pascal user from effectively programming his microprocessor" leads me to believe that you are equating machine programming with effective programming. I think if you consider the programmer's time in coding and debugging, you will find Pascal - even a p-code implementation - to be the more "effective".

I am bringing these problems to your attention to help prevent a situation in which people using your product think they are using Pascal, and try to move programs to a Pascal compiler and blame Pascal for not being your language. To be honest with your customers current and potential, you might choose to refer to HA-PASCAL/I as "a Pascal derivative for microcomputer systems programming" - which it is - rather than "a version of Pascal" - which I don't think it is.

Thank you.

Sincerely,

Ian F. Darwin
University of Toronto Computing Services
10 King's College Road
Toronto, Ontario M5S 1A1

/marlet/tik

CC:
T. Wood
Pascal User Group Newsletter

MELVIN E. CONWAY

July 9, 1981

Mr. Rick Shaw
Pascal Users Group
P.O. Box 88524
Atlanta, GA 30338

Dear Mr. Shaw:

I am interested in joining the Pascal Users Group. Please send information and the necessary materials.

I am an independent contractor who has recently completed a Pascal-in-ROM for the Rockwell AIM 65; I expect Rockwell to release the ROMs this month. The noteworthy thing about this software is that it relates to the user like BASIC: there is no compilation phase requiring external file storage; it talks to the user entirely at the source-language level, including a source-level trace, source-level single-step, and immediate statement execution; and execution is possible right after a source-level change.

The AIM 65 version of the Instant Pascal (my trademark) design implements a substantial subset of the language, including character, string (an extension), real, enumerated, subrange, array, and record data types, as well as all statement forms.

Now that the product is real I am ready to start talking with people who see other uses for this technology, particularly those who are in a position to support its development. Fuller versions of this software for other microcomputers come to mind, as well as more specific tools, such as microprocessor software development systems. Your assistance in getting the word out will be appreciated.

Thanks for your help.

Very truly yours,

Melvin E. Conway

8 BROOK HEAD AVE. BEVERLY, MASS. 01915 U.S.A. PHONE (617) 922-5042

1 DEC 1981

RICK SHAW
PASCAL USER'S GROUP
DIGITAL EQUIPMENT CORPORATION
5775 PEACHTREE DUNWOODY RD.
ATLANTA, GEORGIA 30342

Dear Rick,

I found your address in the back of " Introduction to Pascal for Scientists" by James W. Cooper and so am writing to join the PUG.

I have for the last month owned an APPLE II w/48K, a PASCAL language card, an 80 colum card, two disk drives, an Epson MX-80 printer,and a D.C. Hayes Micromodem. The purpose of all this equipment is to allow use of the PASCAL text editor as a word processor and to communicate my texts with a group of coworkers scattered all across the USA. It has worked well and I now fancy myself as a demon editor, however as a PASCAL programmer, a novice only. A program to select printer options- menu sort of things has been the extent of my programs.

The need for more information is clearly apparent as I have no other programming background to draw from so I am inclosing a few extra dollars(I hope, as I don't know exactly what the fee for joining is) for back issues of PASCAL NEWS- particularly those issues which have information about programs for ..storage and retrieval of files..storage and retrieval of addresses and print out of same..fast Fourier transforms..and most important when writing a letter how do I get the 80 printer to page?

Thanks for whatever time you can spare to help me out.

Regards,

MARVIN SULLIVAN
814 BOCA CIEGA ISLE
ST. PETERSBURG BEACH
FL 33706

TRW

September 24, 1981

Pascal User Group
P. O. Box 888524
Atlanta, GA 30338

Attn: Rick Shaw

Dear Rick,

It was good talking with you last night. I would appreciate you placing the following text in your newsletter:

We would appreciate contact from anyone utilizing Pascal under a VAX/VMS. We are specifically interested in the run-time efficiency of executable code. Any other comments would be appreciated. Please Contact:

Jim Corrigan
TRW, Inc.
5205 Leesburg Pk. (Suite 1106)
Falls Church, VA 22041

(703) 931-2017

Thanks again, Rick.

Yours truly,

Jim Corrigan
TRW, Inc.

JSC/dm

DEFENSE AND SPACE SYSTEMS GROUP OF TRW INC
BEYLINE OFFICE - 5205 LEESBURG PIKE, SUITE 1106, FALLS CHURCH, VIRGINIA 22041 - (703) 931-2000, 931-2017

Three Rivers Computer

Three Rivers Computer Corporation
195 Farmington Avenue
Farmington, Connecticut 06032
203/674-8367

October 28, 1981

Pascal User's Group
P.O. Box 4406
Allentown, PA 18170

RE: Rush Request for Software Package Information

Dear Sir or Madame:

I have the responsibility of identifying "all" of the available software products and packages written in PASCAL. As you are aware, this is a very large task, and I have a very short time to acquire as much information as possible--about two weeks.

I need your help, and the help of as many people as you can contact. There is a benefit to at least some respondents. As you may know, our company produces a high-speed unshared computer (PERQ) which is a Pascal-based machine. We are looking for purchase, contract, OEM, third party and contributed or public domain applications and any other Pascal software. We will be negotiating distribution and license agreements immediately with qualified software sources.

Can you please assist me by: 1) forwarding any present compilations or catalogues you have of available software, to me immediately; 2) passing on this request to any other appropriate parties, by phone, if possible.

I greatly appreciate any information you can provide. Please feel free to contact me anytime at (203) 674-8367. Thank you. I shall look forward to hearing from you.

Kindest regards,

Gary E. Bickford
Sales Support Specialist

GEB/cao

* Devonshire Street, London W1N 2BA Telephone 01-636 6186 Chairman, Professor J C West, CBE Director C Hubbard

The Burleigh Centre
Wellfield Road
HATFIELD
Herts. AL10 0BZ
Tel: Hatfield 74497

Mr N Hughes
Shetlandtel
WALLS
Shetland ZE2 9PF

3rd December 1981

Dear Nick

CET TELESOFTWARE PROJECT

Thank you for your letter of 19th November. I am sorry I have not replied earlier.

Although all our current programs are in BASIC, our format was intended to be independent of language. We would like to distribute programs in other languages, including PASCAL, but on looking into the question, there appear to be a few problems which need to be sorted out first.

Firstly, there are a few characters used in PASCAL not covered by our format recommendations. I hope you have now received your copy of the recommendations and we would, of course, be interested in any comments from members of PUG.

Secondly, as you know, our telesoftware system at present is only available for use with the 380Z. Although PASCAL can be obtained for the 380Z, it will only work on 56K full disc machines with 80 character display.

Thirdly, it appears that very few Computer Assisted Learning programs have so far been written in PASCAL.

In view of these problems, it is likely that in the immediate future only a few people would be able to obtain PASCAL by telesoftware and find it useful. I therefore do not think PASCAL can be one of our first priorities, and we would not consider including programs in our library for a few months until our telesoftware service is fully established.

Thank you for your interest.

Yours sincerely

Chris Knowles

Telesoftware Project Manager, PUG (UK), C/O Shetlandtel, Walls, Shetland, ZE2 9PF. UK.

RESPONSE PLEASE TO: NICK HUGHES, PUG (UK), C/O PUG (UK) on FREEFOL MAILBOX No. 03557/350 { C/O SHETLANDTEL, WALLS, SHETLAND. ZE2 9PF. U.K.

COUNCIL FOR EDUCATIONAL TECHNOLOGY FOR THE UNITED KINGDOM

GENERAL OFFICE * 330 WEST 10th STREET * HIALEAH, FL 33010 * PHONE 305-885-0131

PASCAL USERS GROUP
C/O RICK SHAW
BOX 88524
ATLANTA, GA. 30338

DEAR RICK,

I have spoken with the sales people at Microsoft in an attempt to purchase a copy of their new release of Pascal to run on CP/M. They told me that they were not selling to end users at this, time only to OEM. They also would not reveal the names of any of their OEM users but that if I could locate one maybe one would sell to me. Would you, Mr.Shaw, be able to refer me to any manufacturers who are using Microsoft Pascal and who hopefully would consider selling to an end user.

The main reason I want Microsoft Pascal is the compatibility of their object file format to Digital Research's for link and locate with RMAC assembled files. If you know any other suppliers whose Pascal is compatible to Digital Research's format please let me know.

I also would like to receive some information on the Pascal Users Group.

Thank you for your time. Any help will be appreciated.

Sincerely,

Ted Britton

Nova Robotics 262 Prestige Park Road, East Hartford, CT 06108 (203) 528-7133

September 24, 1981

Rick Shaw
Pascal Users Group
PO Box 888524
Atlanta, GA 30338

Dear Rick:

Nova Robotics is a new user of the Oregon Software OMSI Pascal-2, and we are interested in what the Pascal Users Group has to offer. We have the OMSI Pascal on a PDP 11/34 running RSX-11M V3.2. Enclosed is our check for a one-year subscription.

We are also interested in knowing if any member of the Users Group is developing a Pascal compiler or cross-compiler for Intel's 8086/87. We have talked to Oregon Software. They currently have no plans and suggested we contact the Users Group. Any information you could supply would be appreciated.

Sincerely,

NOVA ROBOTICS LIMITED PARTNERSHIP

Linda J. Phillips

Linda J. Phillips
Manager of Software Engineering

LJP/mh
Enclosures

To the editor:

Members of the Pascal Users Group may obtain a free copy of our new publication, Pascal Market News, by writing to me at the address below. Our publication is commercially slanted towards buyers and makers of Pascal hardware and software. Anyone requesting a free issue should be sure to indicate that he or she is a P. U. G. member.

Ray Jordan
Southwater Corp.
P O Box 5314
Mt. Carmel CT 06518

Rick,

A couple of items:

1. Pascal News continues to be outstanding! You took over a big task from Andy, and have done a super job. Please renew my subscription for three years. (Any possibility of PUG offering a lifetime membership for an appropriate fee?)

2. We are about to begin a large software development project and have choosen Pascal as the implementation language. We are developing a local networking capability for Control Data, IBM, and Honeywell mainframes Users will be able to transfer data files, submit jobs, and route output files among the dissimilar mainframes. The system also includes a global mailbox facility for sending and receiving messages to/from other users.

We chose Pascal because of transportability, structure, and ease of code maintenance. Except for operating system interfaces and machine-dependent routines, the total system will be written in Pascal. It will be developed and maintained as one system, configurable for any of the mainframes.

I would be interested in hearing from any PUG members who have worked on similar large projects in Pascal.

Sincerely,

Mike Burshor
928 Wright Avenue #903
Mt. View, CA 94043

(day) 408-744-5673

EOI ENCOUNTERED.

JET PROPULSION LABORATORY California Institute of Technology • 4800 Oak Grove Drive, Pasadena, California 91103

TO: Distribution

FROM: E. M. Miya

SUBJECT: Suite Report for University of Wisconsin Pascal on Univac 1100

Attached you will find the Validation Suite Report for the UW Pascal compiler on the Univac 1100. Sorry we could not get it to you sooner, it spent some time in our documentation section getting approval.

Please keep us informed about the progress of version 3.0 of the Suite.

Distribution:

R. J. Cichelli
B. Dietrich
A. H. J. Sale
R. Shaw ✓

Telephone (213) 354-6321 Twx 910-588-3580 Telex 910-588-3284

Authored by:
I.E. Johnson, E.N. Miya, S.K. Skedzielewski

Pascal Processor Identification

Computer: Univac 1100/81

Processor: University of Wisconsin Pascal version 3.0 release A

Test Conditions

Testers: I.E. Johnson, E.N. Miya.

Date: April 1980

Validation Suite Version: 2.2

General Introduction to the UW Implementation

The UW Pascal compiler has been developed by Prof. Charles N. Fischer. The first work was done using the P4 compiler from Trondheim, then the NOSC Pascal compiler written by Mike Ball was used, and now all development is done using the UW Pascal compiler.

There are two UW Pascal compilers; one produces relocatable code and has external compilation features, while the other is a "load-and-go" compiler, which is cheaper for small programs. Most tests were run on the "load-and-go" version. Both compilers are 1-pass and do local, but not global optimization. The UW compiler is tenacious and will try to execute a program containing compile-time errors. This causes problems when running the Validation Suite, since programs that are designed to fail at compile time will appear to have executed.

Conformance Tests

Number of Tests Passed: 123

Number of Tests Failed: 16

Details of Failed Tests

Test 6.4.3.5-1 failed on the declaration of an external file of pointers (only internal files of pointers are permitted).

Tests 6.4.3.5-2, 6.4.3.5-3 and 6.9.1-1 failed due to an operating system "feature" which returns extra blanks at the end of a line. This problem affects EOLN detection.

Test 6.5.1-1 failed because the implementation prohibits

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under NASA Contract NAS7-100.

files that contain files.

Tests 6.6.3.1-5 and 6.6.3.4-2 failed because the current version of this implementation prohibits passing standard functions and procedures as parameters.

Test 6.6.5.3-1 failed to assign an already locked tag field in a variant record, but the standard disallows such an assignment! (Error in test?)

Test 6.6.5.4-1 failed to pack because of a subscript out of range. MACC notified.

Test 6.6.6.2-3 failed a nine-digit exp comparison. Univac uses 8 digit floating point.

Test 6.6.6.5-2 failed test of ODD function (error with negative numbers).

Test 6.8.2.4-1 failed because non-local GOTO statements are not allowed by this implementation.

Test 6.8.3.4-1 failed to compile the "dangling else" statement, giving an erroneous syntax error.

Tests 6.9.4-1 and 6.9.4-4 failed do unrecoverable I/O error. Problem referred to MACC.

Test 6.9.4-7 failed to write boolean correctly. UW right-justifies each boolean in its field; the proposed ISO standard requires left-justification.

Extensions

Number of Tests Run: 1

Details of Tests

Test 6.8.3.5-14 shows that an OTHERWISE clause has been implemented in the case stetement.

Deviance Tests

Number of Deviations Correctly Handled: 77

Number of Deviations Incorrectly Handled: 14

Number of Tests Showing True Extensions: 2

Details of Extensions

Test 6.1.5-6 shows that a lower case e may be used in real numbers.

Test 6.1.7-11 shows that a null string is accepted by this implementation.

Details of Incorrect Deviations

Tests 6.2.2-4, 6.3-6, 6.4.1-3 show errors in name scope. Global values of constants are used even though a local definition follows; this should cause a compile-time error.

Tests 6.4.5-3, 6.4.5-5 and 6.4.5-13 show that the implementation considers types that resolve to the same type to be "equivalent" and can be passed interchangeably to a procedure.

Test 6.6.2-5 shows a function declaration without an assignment to the function identifier.

Test 6.8.3.9-4 the for-loop control variable can be modified by a procedure called within the loop. No error found by implementation.

Tests 6.8.3.9-9, 6.8.3.9-13 and 6.8.3.9-14 show that a non-local variable can be used as a for-loop control variable.

Test 6.9.4-9 shows that a negative field width parameter in a write statement is accepted. It is mapped to zero.

Test 6.10-1 shows that the implementation substitutes the default file OUTPUT in the program header. No error message.

Test 6.10-4 shows that the implementation substitutes the existence of the program statement. We know that the compiler searched first but found source text (error correction).

Tests 6.1.8-5 and 6.6.3.1-4 appear to execute; this occured after the error corrector made the obvious changes.

Error Handling

Number of Errors Correctly Detected: 29

Number of Error Not Detected: 17

Details of Errors Not Detected

Tests 6.2.1-7, 6.4.3.3-6, 6.4.3.3-7, 6.4.3.3-8 and 6.4.3.3-12 show that the use of an uninitialized variable is not detected. Variant record fields are not invalidated when the tag changes. 6.4.3.3-12 incorrectly printed "PASS" when it should have printed "ERROR NOT DETECTED".

Test 6.6.2-6 shows the implementation does not detect that a function identifier has not been assigned a value within the function. The function should be undefined. The quality of the test could be improved by writing the value of CIRCLERADIUS.

Test 6.6.5.2-2 again runs into the EOLN problem.

Test 6.6.5.2-6 shows that the implementation fails to detect the change in value of a buffer variable when used as a global variable while its dereferenced value is passed as a value parameter. This sould not cause an error, and none was flagged. However, when the char was changed to a var parameter no error was detected, either.

Test 6.6.5.2-7 shows that the implementation fails to detect the change in a file pointer while the file pointer is in use in a with statement. This is noted in the implementation notes.

Test 6.6.5.3-5 shows the implementation failed to detect a dispose error; but again, the parameter was passed by value, not by reference! (Error in test)

Tests 6.6.5.3-7 and 6.6.5.3-9 show that the implementation failed to detect an error in the use of a pointer variable that was allocated with explicit tag values.

Tests 6.6.6.3-2 and 6.6.6.3-3 show that trunc or round of some real values. 2**36 does not cause a run time error or warning. In those cases, the value returned was negative. Error reported to MACC.

Tests 6.7.2.2-6 and 6.7.2.2-7 show that the implementation failed to detect integer overflow.

Tests 6.8.3.9-5 and 6.8.3.9-6 show that the implementation does not invalidate the value of a for-loop control variable after the execution of the for-loop. Value of the variable is equal to the last value in the loop. These tests could be improved by writing the value of m.

Implementation Defined

Number of Tests Run: 15

Number of Tests Incorrectly Handled: 0

Details of Implementation Definitions

Test 6.4.2.2-7 shows maxint equals 34359738367 (2**35-1).

Test 6.4.3.4-2 shows that a set of char is allowed.

Test 6.4.3.4-4 shows that 144 elements are allowed in a set, and that all ordinals must be >= 0 and <= 143.

Test 6.6.6.1-1 shows that neither declared nor standard functions and procedures (nor Assembler routines) be passed as parameters.

Test 6.6.6.2-11 details a number of machine characteristics such as

XMIN = Smallest Positive Floating Pt # = 1.4693679E-39

XMAX = Largest Positive Floating Pt # = 1.7014118E+38

Tests 6.7.2.3-2 and 6.7.2.3-3 show that boolean expressions are fully evaluated.

Tests 6.8.2.2-1 and 6.8.2.2-2 show that expressions are evaluated before variable selection in assignment statements.

Test 6.9.4-5 shows that the output format for the exponent part of real number is 2 digits. Test 6.9.4-11 shows that the implementation defined default values are:

integers : 12 characters
boolean : 12 characters
reals : 12 characters

Test 6.10-2 shows that a rewrite to the standard file output is not permitted.

Tests 6.11-1, 6.11-2, and 6.11-3 show that the alternative comment delimiter symbols have been implemented; all other alternative symbols and notations have not been implemented. In addition, it is interesting that the compiler's error correction correctly substituted "[" for "(." and ":=" for "%=" as well as a number of faulty substitutions.

Quality Measurement

Number of Tests Runs: 23

Number of Tests Incorrectly Handled: 2

Results of Tests

Test 5.2.2-1 shows that the implementation was unable to distinguish very long identifiers (27 characters). Test 6.1.3-3 shows that the implementation uses up to 20 characters in distinguishing identifiers.

Test 6.1.8-4 shows that the implementation can detect the presence of possible unclosed comments (with a warning). Statements enclosed by such comments are not compiled.

Tests 6.2.1-8, 6.2.1-9 , and 6.5.1-2 show that large lists of declarations may be made in a block (Types, labels, and var).

Test 6.4.3.2-4 attempts to declare an array index range of "integer". The declaration seems to be accepted, but when the array is accessed (All[maxint]), an internal error occurs.

Test 6.4.3.3-9 shows that the variant fields of a record occupy the same space, using the declared order.

Test 6.4.3.4-5 (Warshall's algorithm) took 0.1356 seconds CPU time and 730 unpacked (36-bit) words on a Univac 1100/81.

Test 6.6.1-7 shows that procedures may not be nested to a depth greater than 7 due to implementation restriction. An anomolous error message occurred when the fifteenth procedure declaration was encountered; the message "Logical end of program reached before physical end" was issued at that time, but a message at the end of the program said "parse stack overflow".

Tests 6.6.6.2-6, 6.6.6.2-7, 6.6.6.2-8, 6.6.6.2-9, and 6.6.6.2-10 tested the sqrt, atan, exp, sin/cos, and ln functions. All tests ran, however, typical implmentation answers (which use the Univac standard assembler routines) were slightly smaller than Suite computed. Error typically occurred around the 8th digit (Univac floating-point precision limit).

Test 6.7.2.2-4 The inscrutable message "inconsistent division into negative operands" appears. We think it means that I MOD 2 is NOT equal to I - I div 2 * 2. Problem reported to MACC.

Test 6.8.3.5-2 shows that case constants must be in the same range as the case-index.

Test 6.8.3.5-8 shows that a very large case statement is not permissible (>=256 selections). A semantic stack overflow occurred after 109 labels.

Test 6.8.3.5-18 shows the undefined state is the previous state at the end of the for-loop. The range is checked.

Test 6.8.3.9-20 shows for-loops may be nested to a depth of 6.

Test 6.8.3.10-7 shows with-loops may be nested to a depth of 7.

Test 6.9.4-10 shows that the output buffer is flushed at the end of a program.

Test 6.9.4-14 shows that recursive I/O is permitted using the same file.

Concluding Comments

The general breakdown of errors is as follows:

I/O

These problems are intimately tied to the EXEC 1100 operating system and its penchant to pad blanks on the end of a line. There is no plan to try to correct this problem. Does an external file of pointers make sense!

Changes in the standard

Jensen and Wirth (second edition) was used as the standard for development of this compiler. Since there are discrepencies between it and the ISO proposed standard, several deviations occured. The compiler will be brought into conformance on most of these errors when some standard is adopted.

Restrictions

Some restrictions will be kept, even after a standard is adopted. GOTO's out of procedures will probably never be implemented, but STOP and ABORT statements have been added to the language to alleviate the problem.

Bugs

Several previously unknown bugs were found by running the validation suite. Professor Fischer has been notified, and corrections should be included in the next release of the compilers.

One area that should be emphasized is the clarity of the diagnostics produced by the compiler. All diagnostics are self-explanatory, even to the extent of saying "NOT YOUR FAULT" when an internal compiler error is detected. A complete scalar walk-back is produced whenever a fatal error occurs. The compiler attempts error correction and generally does a very good job of getting the program into execution.

The relocatable compiler has extensive external compilation features. A program compiled using these facilities receives the same compile-time diagnostics as if it were compiled in one piece.

IMPLEMENTATION DESCRIPTION.

DEC-10, DEC-20 (LOTS) PASCAL/PASSGO at LOTS

1. DISTRIBUTOR/IMPLEMENTOR/MAINTAINER:

Distributor/Maintainer:
J. Q. Johnson
LOTS Computer Facility
Stanford University
Stanford, CA 94305 (415)497-3214
Arpanet:
Admin.JQJ@SU-SCORE

Implementor/Maintainer:
Armando R. Rodriguez
Computer Science Department
Stanford University
Stanford, CA 94305

2. MACHINE: Digital Equipment Corp. DEC-10 and DEC-20.

3. SYSTEM CONFIGURATION: DEC TOPS-10, TOPS-20; TEMEX and WAITS nitors, using Concise Command Language (CCL). Uses KA-10 instruction set. Modifications for KI-10 improved inst. set, under development.

4. DISTRIBUTION:
+ Nondisclosure agreement required. See accompanying form.
(We require this with two purposes:
a) To know how many copies are around, and who has them.
b) To prevent the use of our improvements by profit-oriented organizations in products that would later be sold.e)
+ You should provide the transport medium. Methods used until now:
- Through the Arpanet.
- You send us a 9 track tape (no less than 1200 feet, please). Specify density and format desired. (default: 1600 bpi, DUMPER/BACKUP INTERCHANGE ormat).
- You come by and get it on your tape.
+ Distributed on an "as is" basis. Bug reports are encouraged and we will try to fix them and notify you as soon as possible.
+ The compiler is going through a continual, although slow, improvement process. Users, and PUG, will be notified of major new releases and critical bugs.

5. DOCUMENTATION:
+ A modified version of the machine-retrievable manual from the original Hamburg package, as a complement to Jensen & Wirth.
+ A "help" file for online access to the most relevant topics.
+ A NOTES file with comments and hints from local users.
+ An implementation checklist.
+ A description of interesting parts of the internal policies (Packing mechanism, linkage conventions, the symbol table, a complete list of error messages, and a checklist to add predefined procedures).
+ All the documentation machine-retrievable.

6. MAINTENANCE POLICY:
+ We are our own main user: maintenance benefits us first.
+ No guaranteed reply-time.
+ One to four releases a year, for the next two years, at least.
+ Future Plans:
- Support full Standard Pascal
- Optional flagging of use of non-standard features.
- Sets of any size (probably 144-element sets first)
- CHAR going from space to '}'.
- Make the heap a real heap.
- Z0-native version.
- A more friendly user interface: Improvements in the debugger, more and better utility programs, more measurement tools; better error messages.

7. STANDARD:
+ It supports the standard as defined in Jensen & Wirth, except:
- Records, Arrays and Files of Files are not supported.
- Read and Write to non-text Files are not supported.
- Set expressions that contain a range delimited by variables or expressions are not supported.
- The heap works as a stack. Procedure DISPOSE 'pops' the given item and everything else that was created afterwards.
+ Set size is 72 elements, set origin is zero.
+ Type CHAR includes only from space to underbar. No lower case.
+ EXTENSIONS: Type ASCII; functions FIRST, LAST, UPPERBOUND, LOWERBOUND for scalars and arrays, respectively; MIN and MAX; separately compiled procedures; a string manipulation package; LOOP-EXIT construct; OTHERVISE in CASE statements; initialization procedures; DATE, TIME, REALTIME.

8. MEASUREMENTS:
12000+ lines of PASCAL code, 500,000+ chars including comments.
COMPILATION SPEED: around 13,000 chars/sec of CPU time on a 2050.
EXECUTION SPEED: as good as that of the non-optimized FORTRAN compiler.
COMPILATION SPACE: the compiler takes 50k of upper segment, and can work with 16k lower segment.
You receive two compilers (hence the name). They support exactly the same language and features, but one of them (PASSGO) produces the code incore, which saves 25% CPU time and a lot of I/O in the compile-load-and-go sequence. This is ideal for development, and particularly helpful in a student environment.

9. RELIABILITY: Very good. It is very heavily used at LOTS (the program that runs the most, after the editor). Implemented at 30+ sites.

10. DEVELOPMENT METHOD: We started with the Hamburg-76 compiler, distributed by DECUS, which is a very good compiler itself. We have been cleaning bugs, adding missing parts of the standard, and adding features in the last 18 months.

11. LIBRARY SUPPORT AND OTHER FEATURES:
+ Only the essential runtime routines are written in MACRO: most of the library is written in PASCAL.
+ Access to the FORTRAN library support.
+ Access to external FORTRAN and MACRO routines.
+ Separate compilation.
+ Symbolic Post-mortem dump.
+ Interactive runtime source-level debugging package.
+ PCREF, a cross-referencer derived from Hamburg's CROSS.
+ PFORM, a prettyprinter.
+ Statement counts.

Rational Data Systems

Pascal Users Group
c/o Rick Shaw
Digital Equipment Corporation
5775 Peachtreee Dunwoody Road
Atlanta, Georgia 30342

Dear Rick,

Enclosed is a copy of the report of the Validation Suite (2.2) for our Pascal implementations on Data General machines.

Please let me know if you need any further information for publication of this report in Pascal News.

Sincerely,

Douglas R. Kaye
President

DRK/nec

enclosure

245 West 55 Street New York City 10019 212-757-0011 TWX 710-581-6086

Rational Data Systems

PASCAL VALIDATION SUITE REPORT

Processor Identification

Computer: Data General Eclipse
AOS operating system

Processor: Rational Data Systems Pascal
AOS version, release 2.1@

(Implementations for Nova and microNova under RDOS, DOS and MP/OS operating systems are functionally equivalent but were not tested.)

Test Conditions

Tester: Rational Data Systems

Validation Suite Version: 2.2

General Notes

Several tests contained statements of the form "read (f, a[i])", where "f" is a textfile and "a" is a "packed array [1..<n>] of char". In RDS Pascal, the rule that
components of variables of any type designated
packed shall not be used as actual variable
parameters
is applied to "read" and "readln" as well as to user-written procedures and functions. Statements rejected by the compiler were changed to the form "read (f, xx); a[i] := xx", where "xx" is of type "char".

Some tests were not valid because they used 'structural' type compatibility. These were revised accordingly for 'name' type compatibility before running.

Tests passed 128
Tests failed 12 (7 causes)

Details of failed tests:

6.1.2-3: The significance limit is eight characters for both identifiers and reserved words.
6.2.2-3: Type declaration "p = ^node" is incorrectly handled when types named "node" are present both later in same scope and earlier in outer scope.
6.4.3.3-1, 6.4.3.3-3, 6.8.2.1-1: Empty records and empty field lists within record variants are rejected.
6.4.3.3-4: Tagfield "case which: boolean" is rejected when "which" is a known type identifier.
6.5.1-1: A file may not be an element of a record or of an array.
6.6.3.1-5, 6.6.3.4-1, 6.6.3.4-2, 6.6.3.5-1: Procedural and functional parameters are not supported.
6.6.5.3-2: Standard procedure "dispose" is not supported. (Implementation planned for release 2.20).

Details of erroneous tests:

6.1.8-3: Latest draft standard defines "(*" as exactly equivalent to "[", "*)" exactly equivalent to "]".
6.6.5.2-3: Some operating systems distinguish "empty" files (length = 0) from "nonexistent" files (name not known), while others do not.
6.9.4-4: Draft standard requires "write (f, 0.0:6)" to produce floating-point form ("0.0e+00" or similar); suite is testing for fixed-point form ("0.0").
6.9.4-7: Latest draft standard explicitly permits "True" and "False" as well as "TRUE" and "FALSE" when Booleans are written to textfiles.

Tests in which deviations were correctly detected 64
Tests showing true extensions 3
Tests in which erroneous deviations were not detected 26 (13 causes)

Details of tests showing true extensions:

6.1.7-11, 6.4.3.2-5, 6.4.5-11: Type compatibility rules for constant strings weakened to accommodate string-handling extensions.

Details of failed tests:

6.1.2-1: Redeclaration of "nil" permitted.
6.2.1-5: No error message when label is declared but not utilized.
6.2.2-4, 6.3-6, 6.4.1-3: If an identifier is declared in two nested scopes, and there is an erroneous usage of the identifier in the inner scope preceding the definition in the inner scope, the compiler does not detect the error. (Compare conformance test 6.2.2-3.)
6.2.2-7: Nested functions with same name cause erroneous compiletime error message.
6.4.3.3-11: Empty record rejected at compile time.
6.4.5-2: Subranges of same base type treated as identical in parameter/argument case.
6.6.2-5: Function may lack assignment statement.
6.6.3.5-2, 6.6.3.6-2, 6.6.3.6-3, 6.6.3.6-4, 6.6.3.6-5: Procedural and functional parameters not supported.
6.6.6.3-4: Integer arguments to "trunc" and "round" accepted.
6.8.2.4-2, 6.8.2.4-3, 6.8.2.4-4: Tolerates illegal jumps (to nonactivated statement, within structured statement).
6.8.3.9-2, 6.8.3.9-3, 6.8.3.9-4, 6.8.3.9-16: Assignment to control variable of "for" statement allowed.
6.8.3.9-9, 6.8.3.9-14, 6.8.3.9-19: Nonlocal control variable in "for" statement allowed.
6.9.4-9: Nonpositive field width in "write" to textfile allowed.

Details of erroneous tests:

6.1.5-6: Latest draft standard permits both "E" and "e" in real constants.

ERROR HANDLING

Tests in which errors were correctly detected 19
Tests showing true extensions 1
Tests in which errors were not detected 26 (18 causes)

Details of test showing true extension:

6.6.5.2-1: After a file has been opened with "reset", both "get" and "put" operations are allowed. (In fact, both operations are permitted at all times, regardless of how the file was opened.) This extension is provided to permit convenient random processing. RDOS Pascal provides the ability to reposition files with the predeclared procedure "seek (<filename>, <integer expression>)". (Not permitted for files of type "text".)

Details of failed tests:

6.2.1-7, 6.4.3.3-6, 6.4.3.3-8, 6.5.4-1, 6.5.4-2, 6.8.3.9-5, 6.8.3.9-6:
No check is done at runtime for variables with "undefined" (uninitialized, etc.) values.
6.4.3.3-5, 6.4.3.3-7: Storage redefinition is permitted.
6.4.3.3-12: Empty record rejected at compile time.
6.4.6-7, 6.4.6-8, 6.7.2.4-1: No runtime check for illegal set assignments.
6.6.2-6: No runtime check for function that fails to execute assignment statement.
6.6.5.2-6, 6.6.5.2-7: File may be repositioned while buffer is "var" parameter or is record variable of "with" statement.
6.6.5.3-3, 6.6.5.3-4, 6.6.5.3-5, 6.6.5.3-6: Standard procedure "dispose" not supported.
6.6.5.3-7, 6.6.5.3-8, 6.6.5.3-9: Misuse of variable created by variant form of "new" is tolerated.
6.8.3.5-5, 6.8.3.5-6: No runtime error when case-index expression matches none of the case-constants.
6.8.3.9-17: Nested "for" statements may have the same control variable.

IMPLEMENTATION DEFINED

Number of tests run 15

Details of erroneous tests:

6.11-1: Alternate comment delimiters no longer belong to category "implementation-defined"; explicitly required by latest draft standard.
6.11-2: Equivalent symbol for uparrow no longer belongs to category "implementation-defined"; explicitly required by latest draft standard. Equivalent symbols for colon, semicolon, assignment symbol, and square brackets no longer defined; deleted from latest draft standard.
6.11-3: Equivalent symbols for comparison symbols not listed in draft standard.

Details of other tests:

6.4.2.2-7: The value of "maxint" is 32767. (But the value -32768 can be created by writing "-maxint-1" and is not rejected as erroneous.)
6.4.3.4-2: Declaration "set of char" is permitted.
6.4.3.4-4: Implementation permits sets to contain as many as 4080 elements. No set may contain negative elements; e.g. "set of 0..4079" is acceptable, "set of -1..4078" is not. This test brought to light a compiler error; the unacceptable declaration "set of -1..+1" was accepted by the compiler. However, an attempt to insert a negative element into a set (any set) will cause a runtime error. (Fixed in release 2.11).
6.6.6.1-1: Procedural and functional parameters not supported.

6.6.6.2-11: Reals are implemented using Data General's standard single-precision floatingpoint format:
sign: one bit
exponent: 7 bits, excess-64 notation
fraction: 24 bits (6 hexadecimal digits)
All results are normalized (i.e. leftmost hexadecimal digit of fraction is always > 0). However, the range of values that can be read from or written onto textfiles is smaller than the range of values that can be represented internally: conversion to/from ASCII is supported only for values in the range 1.0e-75..1.0e+75. Because this test relies on non-detection of underflow at runtime, it could not be executed without extensive modification. Ultimate results were:

beta	16
t	6
rnd	0
ngrd	1
machep	-5
negep	-6
iexp	7
minexp	-64
maxexp	63
eps	9.53674e-7
epsneg	5.96046e-8
xmin	5.39760e-79
xmax	7.23700e+75

6.7.2.3-2: Boolean expression "a and b" is fully evaluated.
6.7.2.3-3: Boolean expression "a or b" is fully evaluated.
6.8.2.2-1: Selection then evaluation for "a[i] := expr".
6.8.2.2-2: Selection then evaluation for "p^ := expr".
6.9.4-5: Two digits written in an exponent.
6.9.4-11: Default field widths for "write" to textfiles:

integers	variable
Booleans	variable
reals	8 characters

Details of other tests:

5.2.2-1, 6.1.3-3: Significence limit for identifiers is eight characters.
6.1.8-4: No warning message generated when comment extends across several source lines.
6.2.1-8: Accepted 50 type declarations.
6.2.1-9: Accepted declaration and siting of 50 labels.
6.4.3.2-4: Declaration "array [integer] of integer" produced error message "array index may not be of type INTEGER".
6.4.3.3-9: Reverse correlation of fields in record.
6.4.3.4-5: This test was revised to use the RDS "time" extension, which is accurate only to the second. Procedure "Warshallsalgorithm" required 184 bytes of object code, and approximately 5 seconds of elapsed execution time (on a multi-user system).
6.5.1-2: Long declarations allowed.
6.6.1-7: Procedure/function nesting limit is eight.
6.6.6.2-6 (sqrt), 6.6.6.2-7 (arctan), 6.6.6.2-8 (exp), 6.6.6.2-9 (sin & cos), 6.6.6.2-10 (ln): RDS personnel not trained in numerical analysis, unable to interpret results of these tests.
6.8.3.5-2: No warning message when a "case" statement contains an unreachable path.
6.8.3.5-8: Accepted large "case" statement.
6.8.3.9-18: After normal termination (i.e. no "goto") of a "for" loop, the control variable has the value of the limit expression. (After execution of "for i := red to pink do ;", the value of "i" is "pink".)
6.8.3.9-20: Accepted "for" statements nested 15 deep.
6.8.3.10-7: Nesting limit of "with" statements is 12.
6.9.4-10: Textfile output is flushed at end of job when linemarker is omitted. (Note that no linemarker is inserted, however.)

QUALITY

Number of tests run 23

Details of erroneous tests:

6.7.2.2-4: Test of "mod" operator not in conformance with latest draft standard. Caused runtime error message "Non-positive Divisor in MOD Operation".
6.9.4-14: Recursive IO using same file allowed. This test contains a superfluous program parameter which caused the error message "program parameter not declared as file in outermost block". After correction of the error, it ran successfully.

EXTENSIONS

Number of tests run 1

Details:

6.8.3.5-14: The "otherwise" clause in a "case" statement is not supported. (Refer to errorhandling tests 6.8.3.5-5 and 6.8.3.5-6.)

To: Pascal News, c/o Rick Shaw

From: David Intersimone - De Marco-Shatz Corp.

Re: Validation of AlphaPASCAL compiler

Here is a copy of a validation of the AlphaPASCAL compiler. I have given a few comments on the compiler and the validation suite in the validation report.

I have sent a copy of the report to Prof. Sale.

David Intersimone
David Intersimone 3/23/83
De Marco-Shatz Corp.
312 Maple Ave.
Torrance, Ca. 90503
(213) 533-5060

ALPHA MICROSYSTEMS AM-100/T

Pascal Validation Suite Report

Pascal Processor Identification

Computer: Alpha Microsystems AM-100/T

Processor: AlphaPASCAL V2.0

Installation: De Marco Shatz Corporation, Torrance, Ca., USA.

Test Conditions

Tested By: David Intersimone

Date: February / March 1981

Validation Suite Version: 2.2

Report Sent To:

Alpha Microsystems, Software Department, Irvine, Ca., USA.

Pascal News, c/o Rick Shaw, Atlanta, Ga., USA.

Prof. Arthur Sale, Department of Information Science, University of Tasmania, Hobart, Tasmania, Australia.

Note:

'AlphaPASCAL' and 'AM-100/T' are trademarks of Alpha Microsystems, Irvine, Ca., USA.

Conformance Tests:

Total Number of Conformance Tests: 139
Number of Tests Passed: 105
Number of Tests Failed: 34 (19 reasons)

Details of Failed Conformance Tests:

Tests 6.1.2-3, 6.3-1 8-character significance for identifiers.

Tests 6.1.6-1, 6.1.6-2, 6.2.1-1, 6.2.1-2, 6.2.2-5, 6.8.2.4-1, 6.8.3.7-3, 6.8.3.9-8 GOTO statements are not permitted without the (80+) compiler option.

Test 6.2.2-3 The global type for the variable 'node' was used causing a mismatched type in the assignment of ptr^:=true;

Tests 6.4.3.3-1, 6.4.3.3-3 Empty records are not allowed.

Test 6.4.3.5-1 Only type or constant identifiers are allowed for file types.

Tests 6.4.3.5-2, 6.9.1-1 EOLN and EOF are not correctly implemented.

Test 6.5.1-1 The type of record fields and arrays cannot be a FILE type.

Tests 6.6.3.1-5, 6.6.3.4-1, 6.6.3.4-2 6.6.3.5-1 procedures and functions passed as parameters are not allowed.

Test 6.6.5.2-3 failed at runtime with 'invalid filename in RESET'.

Test 6.6.5.2-5 A REWRITE of the file sets EOF false.

Test 6.6.5.3-2 DISPOSE is not implemented. AlphaPASCAL uses MARK and RELEASE to recover memory allocated by NEW.

Test 6.6.5.4-1 PACK and UNPACK are not implemented. AlphaPASCAL automatically unpacks packed data structures.

Test 6.7.1-1 Operator precedence was changed for compatability with other Alpha Micro language processors.

Test 6.8.3.5-4 Crashed the compiler.

Test 6.8.3.9-1 Both expressions in a 'FOR' statement are not evaluated before assignment is done.

Test 6.8.3.9-7 ended up in an infinite loop showing that the test at the last increment caused wraparound(overflow) of the FOR variable.

Test 6.9.3-1 The READLN function is not correctly implemented.

Tests 6.9.4-3, 6.9.4-4, 6.9.5-1 It is illegal to READ into a packed character field.

Test 6.9.4-7 WRITE and WRITELN do not accept a Boolean variable as an argument. Also, as with tests 6.9.4-3 et al, it is illegal to read into a packed character field.

Deviance Tests:

Total Number of Deviance Tests: 94
Number of Deviations Correctly Detected: 55
Number of Tests Not Detecting Erroneous Deviations: 25 (16 reasons)
Number of Tests Showing True Extensions: 2 (2 reasons)
Number of Tests Incorrectly Handled: 12 (6 reasons)

Details of Tests Not Detecting Erroneous Deviations:

Test 6.1.2-1 nil can be used with types other than pointers.

Test 6.1.7-6 Strings can have bounds other that (1..n).

Test 6.1.7-9 Cases 1-4 were accepted. Cases 5-7 rejected.

Tests 6.2.2-4, 6.3-6, 6.4.1-3 Some scope errors are not detected.

Test 6.3-5 Signed constants are allowed in places other than constant declarations.

Test 6.4.3.2-5 Strings can a subrange of other than integers as an index type.

Test 6.4.5-2, 6.4.5-3, 6.4.5-4, 6.4.5-13 Type compatibility is used for variables.

Test 6.4.5-11 Operations on strings with different numbers of components are allowed.

Test 6.6.2-5 Function declarations with no assignment for the function identifier are allowed.

Test 6.6.6.3-4 TRUNC and ROUND will accept integer parameters.

Test 6.7.2.2-9 The unary operator plus(+) can be applied to non-numeric operands.

Tests 6.8.3.9-2, 6.8.3.9-3, 6.8.3.9-4 Assignment can be made to the FOR control variable.

Tests 6.8.3.9-9, 6.8.3.9-14, 6.8.3.9-19 Non-local variables can be used as FOR control variables.

Test 6.8.3.9-16 causes endless loop. FOR control variables can be READ.

Test 6.9.4-9 Field width parameters can be zero and negative. Field widths zero and -1 printed the same as field width 1.

Test 6.10-3 Shows that the standard file OUTPUT can be redefined. Compiled and caused a runtime error.

Details of Tests Showing True Extensions:

Test 6.1.7-11 null strings are allowed.

Test 6.10-1 Default file declarations in the program heading are ignored.

Details of Tests Incorrectly Handled:

Test 6.2.1-4 caused a bad pointer reference error in the compiler.

Test 6.4.3.3-11 Empty records are not allowed.

Test 6.4.5-5 Eight(8) character identifier significance.

Test 6.6.1-6 The procedure call one(c) did not have a semicolon (;) at the end of statement. An error message for the undefined forward procedure was not printed.

Tests 6.6.3.5-2, 6.6.3.6-2, 6.6.3.6-3, 6.6.3.6-4, 6.6.3.6-5 Procedures and functions passed as parameters are not allowed.

Tests 6.8.2.4-2, 6.8.2.4-3, 6.8.2.4-4 GOTO statements are not permitted without the (80+) compiler option.

Error Handling Tests:

Total Number of Error Handling Tests: 46
Number of Errors Correctly Detected: 14
Number of Errors not Detected: 27 (16 reasons)
Number of Tests Incorrectly Handled: 5 (2 reasons)

Details of Errors not Detected:

Test 6.2.1-7 Local variables have values even though they were never assigned.

Tests 6.4.3.3-5, 6.4.3.3-6, 6.4.3.3-7, 6.4.3.3-8 No checking is done on the tag field of variant records.

Tests 6.4.6-7, 6.4.6-8 Bounds checking is not done on set types.

Test 6.6.2-6 Execution of a function without assignment of a value to the function variable is allowed.

Test 6.6.5.2-2 GET when the file is at eof does not cause a runtime error.

Tests 6.6.5.2-6, 6.6.5.2-7 did not cause a runtime error when the file position was changed while the file variable was in use.

Tests 6.6.5.3-7, 6.6.5.3-8, 6.6.5.3-9 No checks are made on pointers when they are assigned using the variant form of NEW.

Test 6.6.6.4-4 SUCC on the last value of an ordinal type does not cause a runtime error.

Test 6.6.6.4-5 PRED on the first value of an ordinal type does not cause a runtime error.

Test 6.6.6.4-7 CHR on a value past the limits of CHAR type does not cause a runtime error.

Test 6.7.2.2-6, 6.7.2.2-7 An error does not occur when the result of a binary integer operation is not -maxint <= 0 <= +maxint.

Test 6.7.2.4-1 Overlapping sets do not cause runtime errors.

Tests 6.8.3.5-5, 6.8.3.5-6 A runtime error does not occur when a CASE statement doesn't contain a constant for the value of the case expression.

Tests 6.8.3.9-5, 6.8.3.9-6 A FOR control variable can be used without an intervening assignment.

Test 6.8.3.9-17 Two nested FOR statements can use the same control variable.

Tests 6.9.2-4, 6.9.2-5 No error occurs when reading characters that don't form a valid integer or real.

Details of Tests Incorrectly Handled:

Test 6.4.3.3-12 Empty records are not allowed.

Tests 6.6.5.3-3, 6.6.5.3-4, 6.6.5.3-5, 6.6.5.3-6 DISPOSE is not implemented.

Implementation Defined Tests:

Total Number of Implementation Defined Tests: 15
Number of Tests Incorrectly Handled: 4 (4 reasons)

Details of Implementation Defined Tests:

Test 6.4.2.2-7 MAXINT is defined as 32767.

Test 6.4.3.4-2 Sets of characters are allowed.

Test 6.4.3.4-4 Set bounds are 0..4095

Tests 6.7.2.3-2, 6.7.2.3-3 Boolean expressions are fully evaluated.

Tests 6.8.2.2-1, 6.8.2.2-2 Variables are selected then evaluated.

Test 6.10-2 A REWRITE on the standard output file is allowed.

Test 6.11-1 Alternate comment delimiters are implemented.

Tests 6.11-2, 6.11-3 Equivalent symbols are not implemented.

Details of Tests Incorrectly Handled:

Test 6.6.6.1-1 Functions are not allowed to be passed as parameters.

Test 6.6.6.2-11 resulted in a floating point runtime error.

Test 6.9.4-5 executed in an endless loop. Output file from the WRITELN statement contained iABC.

Test 6.9.4-11 WRITELN does not allow Boolean variables.

Quality Tests:

Total Number of Quality Tests: 23
Number of Tests Incorrectly Handled: 7 (3 reasons)

Details of Quality Tests:

Tests 5.2.2-1, 6.1.3-3 Eight(8) character identifier significance.

Test 6.1.8-4 Unclosed comments are not detected.

Test 6.2.1-8 Fifty(50) Types were accepted.

Test 6.2.1-9 Fifty(50) LABELS were accepted.

Test 6.4.3.2-4 Gave the compile-time message: 'Array is too large'.

Test 6.4.3.3-9 Exact correlation between variant record fields.

Test 6.5.1-2 Long declaration lists are allowed.

Test 6.6.1-7 Seven(7) procedure/function declarations could be nested. Note: the compiler manual states that the max nesting level is 12.

Test 6.7.2.2-4 DIV by negative operands is implemented and consistent. DIV into negative operands is inconsistent. Quotient=TRUNC(A/B) for negative operands. MOD(A,B) lies in (0,B-1).

Test 6.8.3.5-2 Impossible CASE paths are not detected.

Test 6.8.3.9-18 Range checking is done on a CASE statement after a FOR loop.

Test 6.8.3.9-20 FOR statements can be nested to > fifteen(15) Levels.

Test 6.8.3.10-7 Eleven(11) WITH statements can be nested. The compiler manual states that the maximum nesting of procedures, with-do, and record type descriptions is twelve(12).

Test 6.9.4-10 Output is flushed at end-of-Job.

Test 6.9.4-14 Recursive I/O is allowed.

Details of Tests Incorrectly Handled:

Test 6.4.3.4-5 'Processtime' is not implemented.

Tests 6.6.6.2-6, 6.6.6.2-7, 6.6.6.2-8, 6.6.6.2-9, 6.6.6.2-10 Failed to compile because integer constants must be in the range +-32767, 'e' is not accepted as a substitute for 'E' in real constants, the program blocks where too large for the compiler to handle, and the compiler thought it had hit the end of the program when it hadn't. Note: the compiler manual states that the object code for any procedure or function cannot be larger than 2000 bytes.

Test 6.8.3.5-8 failed to compile after 121 case statement parts because the program block was too large.

Extension Tests:

Total # of Extension Tests: 1

Details of Extension Tests:

Test 6.8.3.5-14 The extension 'OTHERWISE' is not implemented. 'ELSE' is accepted to handle the same function.

Notes about the AlphaPASCAL compiler:

Previous versions of AlphaPASCAL used the UCSD Pascal programming system. The new AlphaPASCAL system consists of a compiler, linker, external library and a run-time package. Text editors are used to create source programs. The compiler generates intermediate files for use by the linker. The linker takes the intermediate files and an external library to create a runnable P-code file.

External procedures and functions can be separately compiled and placed in an external library for future linking with programs. Machine language subroutines can also be written and linked into programs.

AlphaPASCAL run-time uses a virtual memory paging system so there is no size limit on P-code files. The run-time package provides for operator interrupts of program execution allowing program termination, program resumption and a backtrace of all procedures and functions currently active.

Comments on the Validation Suite:

1) Some tests are too large (oriented towards mainframes?). SQRT, ARCTAN, LN, etc. tests (6.6.6.2-6,7,8,9,10) should be broken up. These cause problems with a compiler on smaller machines. Correctness of function should use tests acceptable to large and small computers.

2) How about a new validation section called "Performance"? Would showing the performance of compilation and execution (could be part of the QUALITY tests). Could check to see what(if any) optimization is done.

3) What good is the EXTENSION test and extension tests as part of DEVIANCE? Most deviations are extensions. Isn't the object of the suite to test language standards? All production compilers are going to have extensions. Some extensions will be "standard" in the industry while others will be strictly custom.