PASCAL USERS GROUP

Pascal News

Communications about the Programming Language Pascal by Pascalers

Number 24

- Pascal Standards: Progress Report
- Status Report on Version 3.0
- WRITENUM — A Routine to Output Real Numbers
- TREEPRINT — A Package to Print Trees on Character Printers
- Three Proposals for Extending Pascal
- Announcements

Number
24
JANUARY 83

POLICY: PASCAL NEWS (Jan. 83)

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

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

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

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

- *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 15.5 cm lines!)
- Remember: ALL LETTERS TO US WILL BE PRINTED UNLESS THEY CONTAIN A REQUEST TO THE CONTRARY.
- *Pascal News* is divided into flexible sections:

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

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

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

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

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

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

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

Pascal News

Communications about the Programming Language Pascal by Pascalers

JANUARY 1983 **Number 24**

Hello

This is Pascal News and my name is Charlie Gaffney. Much has happened since I received my March #22-23 Issue. I am the publisher of USUS News. USUS is the UCSD p-System User Society. The p-system was developed to bring Pascal to micro computers. Our USUS News was modeled on Pascal News. We have a lot of information in USUS but it was a chore to read because of bad original and photo copy material used for printing.

I sought a typesetter and found we could typeset and print for only 10% increase in cost. This is a small premium cost to have a readable newsletter. We typeset in August and received many compliments so far.

I thought of our model Pascal News and called Rick Shaw to explain our (USUS) improvement and ask if he needed help.

But Rick had his own story to tell. The work at Pascal Users Group was not performed by a group but by one man, Rick Shaw. He was hard pressed to keep up with the business of PUG.

An offer had been made by the "Journal of Pascal & Ada" to take all pending articles and publish them.

I made a counter offer to maintain PUG as it is under new management. Rick thought that was a nice idea, but the problems would persist and PUG would fail either now or later. After three phone calls Rick decided to let me try.

The News will be typeset and I hope you approve of our new appearance. The articles you submit may be in any format because they will now be typeset. It is possible to enlarge the program listings if they are submitted in a narrow format of 15.5 cm wide.

Business

I have decided to pay a small business to update:

1. the member list
2. new and renew members
3. banking records

Membership costs have gone up but if you pay for two years the third year is free.

Back issues have tied up a great deal of money. We have articles and programs just waiting for you. Buy a set. Buy a complete set. Buy a set for your friends.

A little about me

I am an electrician, and I work for Chevrolet in Parma, Ohio. I have no college education and no formal computer training. My experience with computers involved the purchase of a Western Digital microengine, 16 bit computer. The computer uses p-code as defined by UCSD p-System and directly implements the code without an interpreter. Pascal News and USUS News, and 25 text books, have been my teachers. I thank them and each of you.

Charlie

A Pascal Bibliography

By Tony Heyes
Blind Mobility Research Unit,
Department of Psychology,
University of Nottingham
England

Introduction

The Pascal Bibliography is a package of programs written in standard Pascal and should therefore be easily transported. It enables users to store references and to retrieve them either by AUTHOR name or by KEYWORD; or logical combinations of AUTHORS and KEYWORDS. The bibliography is designed for human use; it uses very explicit prompts.

Design Philosophy

The bibliography consists of a collection of ITEMS. Each ITEM takes the form of:-

One line devoted to AUTHOR or ADDRESSEE names.
Two lines devoted to TITLE or ADDRESS.
Two lines devoted to LOCATION.
DATE ITEM NUMBER.
Two lines devoted to KEYWORDS.

For example:-

HEYES A.D.,FERRIS A.J.,ORLOWSKI R.J.
COMPARISON BETWEEN TWO METHODS OF RESPONSE FOR
AUDITORY LOCALISATION IN THE AZIMUTH PLANE.
J. ACOST. SOC. AMER., 58; 1336-1339

1975 260
DEAFNESS,LOCALISATION,AUDITORY DISPLAYS
STEREOPHONIC SOUNDS,KINAESTHESIS

If ITEMS are addresses the convention is to store the address on the two lines of title.

For example:-

BLOGGS J.B.
Mr.J.B.Bloggs\13 Fishpond Rd.\ Beeston,
Nottingham\ NG7 2RD\ U.K.
Tel 0602-251234

1980 27
ADDRESS,CIRCULATION LIST,XMAS CARD

Note the use of the backslash [\] to indicate the start of a new line. Note also that additional information such as the telephone number can be stored on the location lines. Note, finally, the date has little meaning in this context.

Items may be located by running the program "bibout".

Items may be APPENDED or CHANGED by running the program "bibin".

Both programs are well supplied with prompts and are very simple to use.

Since additions and changes require that the current DICTIONARY be recompiled and this takes time, the actual changes take place during the night. The instructions to implement the changes reside in a PENDING TRAY until the night time run. The user will remain unaware of this slight restriction unless he tries to locate an ITEM during the day on which the ITEM was loaded.

Method of Use

The following assumed the use of the UNIX operating system. Login with your user name, give your password, respond to the first system prompt "%" with "cd bib", ie. change directory to "bib". In answer to the next system prompt, "%", you may select any one of the programs from within the package.

These are:-

a) "bibbin" to enter new items or to change an ITEM.
b) "bibout" to search the bibliography for an ITEM.
c) "outdict" to produce a hard copy of the current DICTIONARY.
d) "cat scratch lpr" to output a hard copy of the SCRATCH FILE.

NEW USERS SHOULD ASK IF THEY MAY HAVE ACCESS TO AN ESTABLISHED BIBLIOGRAPHY AND THEN TRY USING "bibout" TO LOCATE ITEMS OF INTEREST.

To logout respond to the system prompt "%" by typing "control Z".

The Programs

a) "bibin"
The opening prompt allows the selection of one of the following options:-
APPEND

The prompts should be sufficiently explicit, but note:-

(1) Authors and keywords should be separated by commas. Since they are used in the dictionary they should not spill over the end of a line. They can be any length but only the first 20 characters are significant.
(2) The terminal will probably be set to produce lower case letters. The program will automatically convert them to upper case. If you wish to override this, begin each line of text with a backslash [\].
(3) The date must be a single integer e.g. 1980.
(4) If addresses are to be stored use the two title lines, close pack but indicate new lines with a backslash [\].
(5) A personal local storage reference may be kept on the second location line. It should be enclosed in square brackets; e.g. [BM760] means that a copy of this ITEM is in the BM library, entry number 760.

CHANGE

Answer the prompts but please take note of the following:-

1) You must know in advance the ITEM number of the ITEMS you require to change.
2) You have to retrieve the ITEMS from the bibliography so CHANGE is relatively slow; be patient. It saves time, if you are changing more than one ITEM to make the changes in numerical order of ITEM number.
3) You retrieve the ITEM to be changed from the bibliography, the changed ITEM goes into the PENDING TRAY. If you change the same ITEM more than once in a single day only the last version will survive.

SPECIAL FACILITY

This option moves the contents of the SCRATCH file into the PENDING tray. It can be used for moving ITEMS from one bibliography to another. Since SCRATCH is a text file, ITEMS may be changed using an editor and then loaded back into the PENDING tray. (Clever stuff!!).

b) "bibout"

The computer will count the ITEMS in the bibliography and then offer the option of producing a HARD COPY of the dictionary or doing a SEARCH for ITEMS.

SEARCH

You may either search by NUMBER or, more usually by using the DICTIONARY.

You may opt to send the results either to the TERMINAL or to the SCRATCH FILE for subsequent printing.

SEARCH by NUMBER

The search is terminated by asking to search for item number zero [0].

A block of ITEMS may be searched for by asking to search for item number minus one [−1]. You will then be asked for the lowest and the highest item numbers of the block.

SEARCH by DICTIONARY

You will be asked for a word i.e. an AUTHOR name or a KEYWORD. The computer will look this up in the DICTIONARY and list the ITEM numbers of all ITEMS containing this word in their AUTHOR or KEYWORD string. If you are doing a single word search answer the next prompt will a full stop [.], and then the instruction to LOOK UP. If, however, it is a multiple word search give the next word. Once again the corresponding ITEM number list will be printed out.

The answer to the prompt "AND, OR or NOT" enables you to combine the current ITEM number list with the previous ITEM number list. For instance:-

AND Only numbers present in both lists are retained.
OR All numbers from both lists are retained.
NOT Numbers present in the current list are deleted from the previous list.

A new current list is printed out showing the results of the selection. The search sequence may be continued for any number of logical combinations of words. At any time a search for the ITEMS in the current list may be initiated by giving a full stop [.]. After which you may either LOOK UP the selected ITEMS or, if you have made a mistake in your list combinations simply RESTART. There is one special word, namely ***, this word will match all the dictionary.

c) "outdict"

No prompts and no option, simply type "outdict" in answer to the system prompt "%" to obtain a hard copy of the current DICTIONARY.

Note, you must have first prepared a copy of the DICTIONARY by running the appropriate HARD COPY option of "bibout".

d) "opr scratch"

This program is run to obtain the printed output from "bibout", provided the option had been chosen to send the output to the SCRATCH FILE.

No prompts and no options, simply type "opr scratch" in answer to the system prompt "%" to obtain a hard copy of the contents of the SCRATCH FILE.

N.B. If you would like to list the SCRATCH FILE to the terminal to check the contents then run "cat scratch".

Acknowledgements

I gratefully acknowledge the encouragement and support I have received from Roger Henry and Chris Blunsdon.

The bibliography was originally intended for use by the members of the BLIND MOBILITY RESEARCH UNIT it is however available to any members of the Pascal Users Group. Would anyone wishing to take up this offer please contact Tony Heyes to arrange medium of transportation.

NOTES FOR IMPLEMENTORS

The following notes outline the steps the imple-

menter should take in order to establish a new bibliography. After this groundwork, the user can use the shell commands *bibin, bibout,* and *outdict* to build and manipulate the bibliography.

1. The bibliography system requires 6 workfiles named b1 to b6. The recommended practice is for the user to devote a directory to the bibliography, say 'user/bib'. The workfiles can be created easily using the cat command. E.g

   cat > b1 Ẑ

   File b3 requires a link named scratch. This can be created by the command —

   ln b3 scratch

2. b6 is used as a temporary scratch file during the overnight run. It grows to be as large as b1. If there is insufficient room on the user's disc b6 may be coerced on to another disc.
3. The bib directory must contain the following shell commands:-

bibin	Bibin.out b1 b2 b3 b4 b5
bibout	Bibout.out b1 b2 b3 b4 b5
bibupdate	Bibupdate.out b1 b2 b3 b4 b5 b6
outdict	(lpr b4;rm b4; > b4)&

4. Finally, an entry must be made in the UNIX table 'crontab' so that bibupdate will be executed during the night.

```
program Bibin(input,output,bank,dict,scratch,dlist,PendingTray);
(* To ADD, CHANGE or REMOVE items,
 instructions left in a PendingTray file 'pending',
actual changes made by running "Bibupdate.p"  *)
(*  written by Tony Heyes, Blind Mobility Research Unit,
Department of Psychology, The University,
Nottingham, U.K.  *)

label 10;

const    LineLn = 70;
      RowLn = 20;
      HiTag = 10000;
      NonDate = -1066;

type     string = packed array [1..LineLn] of char;
     item = record
               authors,titlel,title2,
               placel,place2 : string;
               date       : integer;
               keyl,key2  : string
            end;
     word = packed array [1..20] of char;
     row = array [1..RowLn] of integer;
     dic = record
              name      : word;
              numbers   : row;
              cont      : boolean
           end;
     TagItem = record
                  tag : integer;
                  entry : item
               end;

var   empty,entry : item;
    bank : file of item;
    PendingTray,TempPendingTray : file of TagItem;
    dlist,scratch  : text;
    dict : file of dic;
    TagEntry : TagItem;
    ch,AppendOption,ChangeOption,MainOption,HelpOption,
      SpecialOption : char; chge : boolean;
    a,n,nn,count : integer;

procedure InlChar (var ch : char);
(* to read the first character of a word typed into the terminal *)
begin
   ch := input^;
   while not (ch in ['A'..'Z','a'..'z']) do
   begin  (* skips along until first character found *)
      get(input);
      if eoln(input)
        then
          begin
            writeln;
            write('ERROR: character required ....  ')
          end;
      ch := input^
   end;
   while not eoln(input) do   (* skips over rest of line *)
      get(input)
end;  (* of InlChar *)

procedure InlInt (var int : integer);
(* to read an integer and not cause a fatal error if a
     character is given *)
var ch : char;
    a,OrdZero : integer;
    NegFound : boolean;
begin
   repeat  (* skips along until integer is found *)
      get(input);
      if eoln(input)
         then
            begin
               writeln;
               write('ERROR: digit required ....  ')
            end;
      ch := input^
   until ch in ['-','+','0'..'9'];
   if ch='-'
      then
         begin
            NegFound := true;
            get(input);
            ch := input^
         end
      else
         begin
            NegFound := false;
            if ch='+'
               then
                  begin
                     get(input);
                     ch := input^
                  end
         end;
   a := 0;
   OrdZero := ord('0');
   repeat
      a := 10*a+ord(ch)-OrdZero;
      get(input);
      ch := input^
   until not (ch in ['0'..'9']);
   while not eoln(input) do  (* skips over rest of line *)
     get(input);
   if NegFound
      then
         int := -a
         else
            int := a
end; (* of InlInt *)

procedure VDUinString(var str : string);
(*  to input from terminal  *)

var i,n : integer;
    ch : char;
    AllCaps : boolean;
begin
   n := 0;
   AllCaps := true;
   repeat
      n := n+1;
      read(ch);
      if (n=1) and (ch=' ')
         then
            n := 0;
      if (n=1) and (ch='\')
         then
            begin  (* defeat automatic shift with '\' *)
               AllCaps := false;
               n := 0
            end;
      if n<>0
         then
            begin
               if AllCaps
                  then
                     if ch in ['a'..'z']
```

```
                    then
                       ch := chr(ord(ch)-32);
              str[n] := ch
            end
   until eoln(input);
   for i:=n+1 to LineLn do
      str[i] := ' '
end; (* of VDUinString *)

procedure ScratchInStr(var str : string);
(* input from file scratch  *)
var n,i : integer;
    ch : char;
begin
   if not eof(scratch)
      then
         begin
            n := 0;
            repeat
               read(scratch,ch);
            until (ch=':') or (eof(scratch));
            while (not eoln(scratch)) do
               begin
                  read(scratch,ch);
                  n := n+1;
                  str[n] := ch;
               end;
             if n+1<=LineLn
               then
                  for i:=n+1 to LineLn do
                     str[i] := ' ';
         end
 end;   (* of ScratchInStr  *)

 function ScratHoldsItems : boolean;
 (* to inspect the SCRATCH FILE and check that ITEMS are complete *)
   var count,LineNo : integer;
       FaultFound,HeadingError,NegFound : boolean;
   procedure CheckLine;
     var CharCount : integer;
         LineTooLong,BadLine : boolean;
     begin
       LineNo := LineNo + 1;
       CharCount := 1;
       BadLine := false;
       LineTooLong := false;
       get(scratch);
       while (not eoln(scratch)) and (CharCount < LineLn + 9 ) do
         begin
           get(scratch);
           CharCount := CharCount + 1;
           if (CharCount = 9) and (scratch^ <> ':') then
               BadLine := true;
         end;
       if CharCount < 9 then BadLine := true;
       while not eoln(scratch) do
         begin
           get(scratch);
           if scratch^ <> ' ' then LineTooLong := true
         end;
       if BadLine then
         begin
           FaultFound := true;
           writeln('Line',LineNo : 4,'  bad line '':'' missing.')
         end;
       if LineTooLong then
         begin
           FaultFound := true;
           writeln('Line',LineNo : 4,'  overflow.')
         end
     end;  (* of CheckLine *)
   begin
     LineNo := 0;
     HeadingError := false;
     FaultFound := false;
     NegFound := false;
     writeln;
     writeln('SCRATCH FILE CHECK in progress.');

   writeln;
   while not eof(scratch) and not HeadingError do
     begin
       repeat
         get(scratch);
         if not eof(scratch) then
         if eoln(scratch) then LineNo := LineNo + 1
       until (eof(scratch)) or (scratch^ = '-');
       if scratch^ = '-' then NegFound := true;
       LineNo := LineNo + 1;
       if eof(scratch) then
         begin
           if not NegFound then  (* no ITEMS present *)
             begin
               HeadingError := true;
               writeln('SCRATCH does not contain ITEMS.')
             end
         end
         else
         begin
           while not eoln(scratch) do get(scratch);
           for count := 1 to 5 do CheckLine;
           LineNo := LineNo + 1;
           get(scratch);
           while (not eoln(scratch)) and
                 not (scratch^ in ['1'..'9']) do get(scratch);
           while (not eoln(scratch)) and
                 not (scratch^ = ' ') do get(scratch);
           while (not eoln(scratch)) and
                 not (scratch^ in ['1'..'9']) do get(scratch);
           if eoln(scratch) then
             begin (* two numbers not present *)
               FaultFound := true;
               writeln('Line',LineNo : 4,
                       '  two integers not found.')
             end
             else
             while not eoln(scratch) do get(scratch);
           for count := 1 to 2 do CheckLine;
         end;
       if FaultFound then
         begin
           writeln;
           writeln('Errors in SCRATCH use editor to correct,
                    then try again.');
           writeln;
           ScratHoldsItems := false
         end
         else if not HeadingError then ScratHoldsItems := true;
       reset(scratch)
   end; (* of ScratHoldsItems *)

procedure empt; (* to empty an ITEM *)
var NoChar : string;
    a  : integer;
    ch : char;
begin
   NoChar[1] := ' ';
   NoChar[2] := ' ';
   NoChar[3] := ' ';
   for a:=4 to LineLn do
      NoChar[a] := '.';
   with empty do
      begin
         authors := NoChar;
         title1 := NoChar;
         title2 := NoChar;
         place1 := NoChar;
         place2 := NoChar;
         date := NonDate;
         key1 := NoChar;
         key2 := NoChar
      end;
   for a:=2 to 9 do
      begin
         case a of
            2: ch := '.';
            3: ch := 'e';
            4: ch := 'm';
            5: ch := 'p';
            6: ch := 't';
            7: ch := 'y';
            8: ch := ' ';
            9: ch := ' '
         end;  (* of case *)
         empty.authors[a] := ch
      end
end; (* of empt *)

procedure OutRecord(entry : item; n : integer);
(*  to write to the terminal *)
var a : integer;
begin
   for a:=1 to 7 do
      write('---------I');
   writeln;
   with entry do
      begin
         writeln(authors);
         writeln(title1);
         writeln(title2);
```

```
        writeln(place1);
        writeln(place2);
        writeln(date:8,'      Item number :',n :5);
        writeln(key1);
        writeln(key2)
      end
end;  (* of OutRecord *)

procedure GetReference(n : integer);
(* to count through bank to find an ITEM *)
begin
   if n<count
      then
         begin
            reset(bank);
            count := 1
         end;
   while (count < n) and (not eof(bank)) do
      begin
         count := count+1;
         get(bank)
      end;
   if eof(bank)
      then
         begin
            writeln;
            writeln(' You have only got',count -1,' Items.');
            writeln;
            goto 10
         end
      else
         OutRecord(bank^,n)
end; (* of GetReference *)

procedure change(var entry : item; m : integer);
(* to change the mth. ITEM *)
var  line : integer;
    DMOption,LineOption  :  char;
    str : string;
begin
   writeln;
   writeln;
   repeat
      write('Do you wish to DELETE or MODIFY ....      ');
      InlChar(DMOption)
   until DMOption in ['D','d','M','m'];
   if DMOption in ['D','d']
      then
         begin
            empt;
            entry := empty
         end
      else
         begin
            writeln;
            writeln('You may REPLACE a line,');
            writeln('move to the NEXT line,');
            writeln('or SKIP to the end of the item. ');
            writeln;
            line := 0;
            repeat
               line := line+1;

with entry do
   case line of
      1: str := authors;
      2: str := title1;
      3: str := title2;
      4: str := place1;
      5: str := place2;
      6: ;
      7: str := key1;
      8: str := key2
   end;  (* of case *)
if line<>6
   then
      begin
         writeln;
         writeln(str);
         writeln(output);
         repeat
            write('REPLACE, NEXT line or SKIP to end ....  ');
            InlChar(LineOption)
         until LineOption in ['R','r','N','n','S','s'];
         writeln;
         if LineOption in ['R','r']
            then
               begin
                writeln('Type replacement line  :');
                writeln;
                VDUinString(str);
                with entry do
                case line of
                 1: authors := str;
                 2: title1 := str;
                 3: title2 := str;
                 4: place1 := str;
                 5: place2 := str;
                 7: key1 := str;
                 8: key2 := str
                end; (* of case *)
               end
      end
   else
      begin
         writeln('Date   ',entry.date :4);
         writeln;
         repeat
            write('REPLACE, NEXT line or SKIP to end ....  ');
            InlChar(LineOption)
         until LineOption in ['R','r','N','n','S','s'];
         if LineOption in ['R','r']
            then
               begin
                writeln('Type replacement date ');
                write(':  ');
                InlInt(entry.date)
                                end;
                        end
           until ((line=8) or (LineOption in ['S','s']));
         end;
   writeln;
   writeln('Modified item reads :    ');
   writeln;
   OutRecord(entry,m);
   writeln;
end; (* of change *)

begin (* MAIN PROGRAM *)
   count := HiTag;
   n := 1;
   reset(PendingTray);
   rewrite(TempPendingTray);
   while not eof(PendingTray) do
      begin  (* copy down existing contents of file
                'PendingTray' *)
         TempPendingTray^ := PendingTray^;
         put(TempPendingTray);
         get(PendingTray)
      end;
   rewrite(PendingTray);
   reset(TempPendingTray);
   while not eof(TempPendingTray) do
      begin  (* copy back 'PendingTray' and count contents *)
         PendingTray^ := TempPendingTray^;
         put(PendingTray);
         get(TempPendingTray);
         n := n+1
      end;
   rewrite(TempPendingTray);

   repeat
      writeln;
      repeat
         write('Do you wish to APPEND, to CHANGE, ');
         writeln('to use the SPECIAL facility, ');
         write('or to FINISH ....  ');
         InlChar(MainOption)
      until MainOption in ['A','a','C','c','S','s','F','f'];

(*  MainOption= S is a special facility,
used  for loading from 'scratch' to 'PendingTray' *)

      case MainOption of
         'A','a':    (* TO APPEND *)
                  begin
                     writeln;
                     repeat
                        write('Do you need help
                              [YES or NO] ....  ');
                        InlChar(HelpOption)
                     until HelpOption in ['Y','y','N','n'];
                     if HelpOption in ['Y','y']
                then
                   begin
writeln;
writeln('NOTES.');
write('(a) Authors and keywords separated');
writeln(' by a comma ",".');
write('(b) To remove the automatic conversion to ');
writeln('upper case letters');
write('    begin a line of text with');
writeln(' a backslash "\".');
```

```
write('(c) Date must be a single integer number');
writeln(' eg. 1980.');
write('(d) If addresses are to be entered use the two');
writeln(' title lines;');
write('    close pack but indicate new');
writeln(' lines with a backslash "\".');
write('(e) A personal local storage reference');
writeln(' may be kept on the 2nd. location line');
write('    but should be enclosed in square brackets;');
writeln(' for example: [EM360].')
                end;
            repeat
               writeln;
               writeln('New item:- ');
               writeln;
               for a:=1 to 7 do
                  write('---------I');
               writeln;
               with entry do
                  begin
      writeln( 'Line of author names, or name of addressee :' );
                     VDUinString(authors);
                     writeln('First line of title or address :');
                     VDUinString(title1);
                     writeln('Second line of title or address :');
                     VDUinString(title2);
                     writeln('First line of reference location :');
                     VDUinString(place1);
                     writeln('Second line of reference location :');
                     VDUinString(place2);
                     writeln('Date - just the year -    :');
                     InlInt(date);
                     writeln('First line of keywords :');
                     VDUinString(key1);
                     writeln('Second line of keywords :');
                     VDUinString(key2);
                  end;
               writeln;
               OutRecord(entry,n);
               repeat
                  writeln;
                  repeat
write( 'Do you wish to make a change [YES or NO] ....      ' );
                     InlChar(ChangeOption)
                       until ChangeOption in ['Y','y','N','n'];
                       if ChangeOption in ['Y','y']
                          then
                           change(entry,n)
                     until ChangeOption in ['N','n'];
                     if entry.date <> NonDate
                        then
                           begin
                            TagEntry.tag := HiTag;
                            TagEntry.entry := entry;
                            PendingTray^ := TagEntry;
                            put(PendingTray);
                            n := n+1
                           end
                        else
                           begin
                            writeln;
                            writeln('Item withdrawn.');
                            writeln
                           end;
                     writeln;
                     repeat
    write( 'Do you wish to append more items [YES or NO] ....  ' );
                        InlChar(AppendOption)
                      until AppendOption in ['Y','y','N','n'];
                   until AppendOption in ['N','n']
               end;  (* of Append option *)

     'C','c':   (* TO CHANGE *)
               begin;
                  writeln;
                  repeat
                     write('Do you need help [YES or NO] ....  ');
                     InlChar(HelpOption)
                  until HelpOption in ['Y','y','N','n'];
                  if HelpOption in ['Y','y']
                     then
                        begin
writeln;
writeln( 'You MUST know the ITEM NUMBERS of the ITEMS you wish to
        change.' );
writeln( 'If you do not, leave this program and run "bibout" to
        find them.' );
writeln( 'Changes do not take place immediately, they stay in the
        PENDING' );
writeln('tray until the "update" program is run.');
writeln( 'If an ITEM is changed more than once only the last
        version survives.'
                        end;
                  repeat
                     10: writeln;
                     chge := false;
               writeln('Type 0 if no ITEM needs changing, otherwise
                     type');
               write('the ITEM number...  ');
                     InlInt(nn);
                     if nn<0
                        then
                           begin
                  writeln;
                  writeln('No negative numbered ITEMS')
                end;
             if nn > 0
                then
                   begin
                    writeln;
                    GetReference(nn);
                    if not eof(bank)
                     then
                      begin
                       entry := bank^;
                       repeat
                        writeln;
                        repeat
 write('Do you wish to change this item [YES or NO] ....  ');
                          InlChar(ChangeOption)
                         until ChangeOption in ['Y','y','N','n'];
                         if ChangeOption in ['Y','y']
                          then
                           begin
                            change(entry,nn);
                            chge := true
                           end
                       until ChangeOption in ['N','n'];
                       TagEntry.tag := nn;
                       TagEntry.entry := entry;
                       if chge
                        then
                         begin
                          PendingTray^ := TagEntry;
                          put(PendingTray);
                          n := n+1
                         end
                      end
                   end;
                writeln;
             until nn = 0
          end; (* of Change option *)

'S','s':  (* To move from text file 'scratch' to 'PendingTray' *)
          begin
             writeln;
             write('This option moves the contents of the ');
             writeln('SCRATCH file into the PENDING tray.');
             write('It can be used to copy selected ITEMS from one');
             writeln(' bibliography to another.');
             write('OR, it can be used to reinstate ITEMS ');
             writeln('which have been changed by the editor.');
             writeln;
             repeat
               writeln;
  write('Do you wish these items to be APPENDED, REINSTATED or
          NO ACTION ....  ');
                     InlChar(SpecialOption)
            until SpecialOption in ['A','a','N','n','R','r'];
            if SpecialOption in ['A','a','R','r']
               then
                  begin
                     reset(scratch);
                     writeln;
                    (* now check that scratch holds ITEMS in
                       the correct form  *)
                     if (not eof(scratch)) and
                       ScratHoldsItems
                         then
                         begin
                           while not eof(scratch) do
                            begin
                             with entry do
                              begin
                               ScratchInStr(authors);
                               ScratchInStr(title1);
                               ScratchInStr(title2);
                               ScratchInStr(place1);
                               ScratchInStr(place2);
                               read(scratch,date);
```

```
                                repeat
                                  read(scratch,ch)
                                until ch = ':';
                                readln(scratch,TagEntry.tag);
        writeln(n,' Dated ',date,'  Item number ',TagEntry.tag);
                                ScratchInStr(key1);
                                ScratchInStr(key2);
                               end;
                              if SpecialOption in ['A','a'] then
                              TagEntry.tag := HiTag;
                              TagEntry.entry := entry;
                              PendingTray^ := TagEntry;
                              put(PendingTray);
                              n := n+1;
                              if not eof(scratch)
                               then
                                get(scratch)
                             end;
                            rewrite(scratch)
                          end
                     end
            end;  (* of Special option *)

    'F','f': begin
               writeln;
               writeln('Number of ITEMS now in Pending
                        Tray =',n-1 :5);
               writeln
             end
     end  (* of case "MainOption" *)
   until MainOption in ['F','f']
end.   (* end of program Bibin.p  *)
```

```
program Bibout(input,output,bank,dict,scratch,dlist,PendingTray);
(* To call down items from the bibliography *)
(*  written by Tony Heyes, Blind Mobility Research Unit,
Department of Psychology, The University,
Nottingham, U.K..  *)

label 10;

co       LineLn = 70;
      RowLn = 20;
      HiTag = 10000;
      LinesPerPage = 64 ;
      VDULinesPerPage = 24;

type     string = packed array [1..LineLn] of char;
     item = record
              authors,title1,title2,
              place1,place2 : string;
              date       : integer;
              key1,key2  : string
            end;
     word = packed array [1..20] of char;
     row = array [1..RowLn] of integer;
     dic = record
             name      : word;
             numbers   : row;
             cont      : boolean
           end;
     link = ^DicLine;
     DicLine = record
                 val  : integer;
                 next : link
               end;

var FileAssigned : boolean;
    bank,PendingTray : file of item;
    dlist,AddressFile,scratch  : text;
    dict : file of dic;
    FirstLink,SecondLink,ThirdLink,ptl,here : link;
    low,high,n,NumSoFar,
    LineNo,AddLineNo,count,TopItem,NFromDict,NumW : integer;
    device,FileStyle,MainOpt,NDOption,LogicAction : char;

procedure InlChar (var ch : char);
(* to read the first character of a word typed into the terminal *)
begin
    h := input^;
    ile not (ch in ['A'..'Z','a'..'z']) do
     begin
        (* skips along until first character found *)
      get(input);
      if eoln(input)
         then
            begin
               writeln;
               write('ERROR: character required ....  ')
            end;
        ch := input^
    end;
    while not eoln(input) do   (* skips over rest of line *)
       get(input)
end;  (* of InlChar *)

procedure InlInt (var f : text; var int : integer);
(* to read an integer and not cause a fatal error if a character
   is given *)

var ch : char;
    a,OrdZero : integer;
    NegFound : boolean;
begin
   repeat  (* skips along until integer is found *)
      get(f);
      if eoln(f)
         then
            begin
               writeln;
               write('ERROR: digit required ....  ')
            end;
      ch := f^
   until ch in ['-','+','0'..'9'];
   if ch='-'
      then
         begin
            NegFound := true;
            get(f);
            ch := f^
         end
      else
         begin
            NegFound := false;
            if ch='+'
               then
                  begin
                     get(f);
                     ch := f^
                  end
         end;
   a := 0;
   OrdZero := ord('0');
   repeat
      a := 10*a+ord(ch)-OrdZero;
      get(f);
      ch := f^
   until not (ch in ['0'..'9']);
   while not eoln(f) do  (* skips over rest of line *)
      get(f);
   if NegFound
      then
           int := -a
        else
           int := a
end; (* of InlInt *)

procedure SkipToEndOfPage(PageLines : integer;
                          var where : text);
begin
   while LineNo < PageLines do
      begin
         writeln(where);
         LineNo := LineNo+1
      end;
   LineNo := 0
end;  (* of SkipToEndOfPage *)

procedure GetRef(n : integer; destination : char);

var a,CharCount,LineInQuestion,NOfCommas,WordLength : integer;
    line : string;
    DoubleSpace,InBrackets,KeepNextCap,
    something,KeepAllCaps,woops : boolean;
    ch,LastCh : char;
begin
   if n<count
      then
         begin
            reset(bank);
            count := 1
         end;
   while (count < n) and (not eof(bank)) do
      begin
         count := count+1;
         get(bank)
      end;
   if eof(bank)
```

```
then
   begin
      writeln;
      writeln(' You have only got',count -1,' Items.');
      writeln;
      goto 10
   end
else
   with bank^ do
      begin
         case destination of
            'T','t': (* Output to terminal *)
         begin
            if (VDULinesPerPage-LineNo < 9)
             then
              SkipToEndOfPage(VDULinesPerPage,output);
            for a:=1 to 7 do
                write('---------I');
                writeln;
                writeln(authors);
                writeln(title1);
                writeln(title2);
                writeln(place1);
                writeln(place2);
                writeln(date:8,'      Item number :',n :5);
                writeln(key1);
                writeln(key2);
                LineNo := LineNo + 9
            end;  (* of 'T' *)
   'I','i': (* Output to scratch file *)
            begin
               if LinesPerPage-LineNo < 9
                then
                 SkipToEndOfPage(LinesPerPage,scratch);
               for a:=1 to 7 do
                write(scratch,'---------I');
               writeln(scratch,'---------');
               writeln(scratch,'Names   :',authors);
               writeln(scratch,'Details :',title1);
               writeln(scratch,'        :',title2);
               writeln(scratch,'        :',place1);
               writeln(scratch,'        :',place2);
               writeln(scratch,date:14,'     Item number:',n :5);
               writeln(scratch,'Keywords:',key1);
               writeln(scratch,'        :',key2);
               LineNo := LineNo + 9
            end;  (* of 'I' *)
   'E','e': (* Output to scratch file in envelope label format.
               Only for addresses. *)
            begin
               writeln(AddressFile);
               AddLineNo := AddLineNo +1;
               woops := true;
               for LineInQuestion:=1 to 2 do
                begin
                 DoubleSpace := false;
                 LastCh := ':'; (* initail value *)
                 CharCount := 0;
                 writeln(AddressFile);
                 AddLineNo := AddLineNo +1;
                 write(AddressFile,'       ');
                 if LineInQuestion=1
                  then
                   line := title1
                  else
                   line := title2;
                 while (CharCount<LineLn) and not DoubleSpace do
                  begin
                   CharCount := CharCount+1;
                   ch := line[CharCount];
                   if ch='\'
                    then
                        begin
                         woops := false;
                         writeln(AddressFile);
                         AddLineNo := AddLineNo +1;
                         write(AddressFile,'       ')
                        end
                       else
                        write(AddressFile,ch);
                      DoubleSpace := (ch=' ') and (LastCh=' ');
                      LastCh := ch
                    end
                  end;
                 while (AddLineNo mod 8) <> 0 do
                  begin
                   writeln(AddressFile);
                   AddLineNo := AddLineNo + 1
                  end;
                 if woops
                  then
                   begin
                        writeln;
                        writeln;
write('An attempt to output a reference');
                        writeln(' in address format.');
                        writeln;
                        writeln;
write( 'Addresses must be close-packed on the two' );
                        writeln(' title lines.');
writeln( 'Use the backslash "\" as line separator.' );
                        writeln;
                        rewrite(scratch);
                        FileAssigned := false;
                        goto 10
                       end
                  end;  (* of 'E' *)
          'R','r': (* Output in format for wordprocessor NROFF *)
                  begin  (* firstly the author line *)
                     writeln(scratch,'.nr');
                     (* this is an NROFF macro *)
                     write(scratch,'\:');
                                    (* bold lettering command *)
                    DoubleSpace := false;
                    KeepAllCaps := false;
                    woops := false;
                    LastCh := ':';       (* initial value *)
                    CharCount := 0;
                    NOfCommas := 0;
                    if authors[1]='\'
                     then
                      begin
                       KeepAllCaps := true;
                       CharCount := CharCount+1
                      end;
                    while (CharCount<LineLn)
                                          and not DoubleSpace do
                     begin
                   CharCount := CharCount+1;
                   ch := authors[CharCount];
                   if ch=','
                    then
                     NOfCommas := NOfCommas+1;
                   DoubleSpace := (ch=' ') and (LastCh=' ');
                   LastCh := ch
                  end;
                 DoubleSpace := false;
                 LastCh := ':';
                 CharCount := 0;
                 while (CharCount<LineLn) and not DoubleSpace do
                  begin
                   CharCount := CharCount+1;
                   ch := authors[CharCount];
                   if (ch in ['A'..'Z']) and (LastCh in ['A'..'Z'])
                      and not KeepAllCaps
                    then
                     write(scratch,chr((ord(ch)+32)))
                    else
                     if ch=','
                      then
                       begin
                        if NOfCommas=1
                         then
                          write(scratch,' & ')
                         else
                          write(scratch,', ');
                        NOfCommas := NOfCommas-1
                       end
                      else
                       write(scratch,ch);
                   DoubleSpace := (ch=' ') and (LastCh=' ');
                   LastCh := ch
                  end;
                 writeln(scratch,'(',date : 4,')\:');
                 for LineInQuestion :=1 to 4 do
                  begin       (* title and place lines *)
                   KeepNextCap := true;
                   KeepAllCaps := false;
                   case LineInQuestion of
                    1: line := title1;
                    2: begin
                        line := title2;
                        KeepNextCap := false
                       end;
                    3: line := place1;
                    4: begin
                        line := place2;
                        CharCount := 0;
                        InBrackets := false;
                        repeat
                         CharCount := CharCount+1;
                         if line[CharCount]='['
                          then
```

```
        InBrackets := true;
       if InBrackets
        then
         if line[CharCount]=']'
          then
           begin
            line[CharCount] := ' ';
            InBrackets := false
           end;
       if InBrackets
        then
         line[CharCount] := ' '
      until CharCount=LineLn
     end
end; (* of case LineInQuestion *)
CharCount := LineLn;
repeat
 CharCount := CharCount-1
until (CharCount=1) or (line[CharCount]<>' ');
if CharCount<LineLn
 then
  line[CharCount+1] := '!'; (* a silly character '
  (* placed at the end of the character stirng *)
WordLength := 0;
if CharCount>1
 then
  repeat
   CharCount := CharCount-1;
   if line[CharCount]<>' '
    then
     begin
      if line[CharCount] in ['A'..'Z']
       then
        WordLength := WordLength+1
     end
    else
     begin
      if not (WordLength in [2,3])
       then
        line[CharCount] := '~';
        (* another silly char fills up spaces
         before words which keep caps. *)
       WordLength := 0
     end
  until CharCount=1;
CharCount := 0;
something := false;
if line[1]='\'
 then
  begin
   KeepAllCaps := true;
   CharCount := CharCount+1
  end;
   ch := ':';  (* initial value *)
    while (CharCount < LineLn) and
          (line[CharCount+1] <> '!') do
    begin
     CharCount := CharCount+1;
     LastCh := ch;
     ch := line[CharCount];
     if not ((LastCh in ['~',' ']) and
        (ch in ['~',' ']))
      then
       begin
        if (ch in ['A'..'Z']) and not KeepNextCap
         then
          ch := chr((ord(ch)+32));
        if ch in ['A'..'Z']
         then
          KeepNextCap := false;
        if ch='\'
         then
          woops := true; (* its an address *)
        if ch='~'
         then
          begin
           ch := ' ';
           if (LineInQuestion in [3,4])
            then
             KeepNextCap := true
          end;
        if (ch in ['1'..'9'])
         then
          KeepNextCap := false;
        if (ch<>' ') and (ch<>'!')
         then
          something := true;
        if something
         then
          write(scratch,ch)
       end
      end;
     if something
      then
       writeln(scratch)
    end;
   if woops
    then
     begin
      writeln;
      writeln;
      write('An attempt to output addresses in');
      writeln(' reference format.');
      writeln;
      writeln;
      rewrite(AddressFile);
      FileAssigned := false;
                       goto 10
                      end
                end   (* of 'R' *)
      end   (* of case destination *)
    end
end; (* of GetRef *)

procedure ReWind(var ptr : link);

var  p,q,pt : link;
begin
   p := ptr;
   pt := nil;
   while p<>nil do
      begin
         new(q);
         q^.val := p^.val;
         q^.next := pt;
         pt := q;
         p := p^.next
      end;
   ptr := pt
end;  (* of ReWind  *)

procedure GetDict(m : integer; var ptr : link);

var       a : integer;
    p : link;
    OldEntry : dic;
    more : boolean;
begin
   if m < HiTag
      then
         begin
            reset(dict);
            a := 1;
            while a<m do
               begin
                  OldEntry := dict^;
                  get(dict);
                  if OldEntry.cont=false
                     then
                        a := a+1
               end;
            writeln;
            writeln(dict^.name);
            ptr := nil;
            repeat
               for a:=1 to RowLn do
                  if dict^.numbers[a]<>0
                     then
                        begin
                           new(p);
                           p^.val := dict^.numbers[a];
                           p^.next := ptr;
                           ptr := p
                        end;
               more := dict^.cont;
               get(dict);
            until not more;
            ReWind(ptr)
         end
      else
         begin
            ptr := nil;
            for a:=TopItem downto 1 do
               begin
                  new(p);
                  p^.val := a;
                  p^.next := ptr;
                  ptr := p
               end
         end
end;  (* of GetDict *)
```

```
procedure join(var p1 :link; p2 : link; which : char);

var       continue : boolean;
    q,qp,pt1,pt2,pt3 : link;
begin
   pt1 := p1;
   pt2 := p2;
   continue := (pt1<>nil) and (pt2<>nil);
   qp := nil;
   case which of
      'A','a':        (* AND *)
               begin
                  while continue do
                     begin
                        if pt1^.val>pt2^.val
                           then
                              begin
                               pt3 := pt1;
                               pt1 := pt2;
                               pt2 := pt3
                              end;
                        if pt2^.val>pt1^.val
                           then
                              begin
                               pt1 := pt1^.next;
                               continue := pt1<>nil
                              end
                           else
                              if pt1^.val=pt2^.val
                               then
                                begin
                                 new(q);
                                 q^.val := pt1^.val;
                                 q^.next := qp;
                                 qp := q;
                                 pt1 := pt1^.next;
                                 pt2 := pt2^.next;
                                 continue := (pt1<>nil) and
                                                    (pt2<>nil)
                                end
                     end
               end;  (* of AND *)
      'O','o':         (* OR *)
               begin
                  begin
                     while continue do
                        begin
                           if pt1^.val>pt2^.val
                              then
                               begin
                                pt3 := pt1;
                                pt1 := pt2;
                                pt2 := pt3
                               end;
                           if pt1^.val<pt2^.val
                              then
                               begin
                                new(q);
                                q^.val := pt1^.val;
                                q^.next := qp;
                                qp := q;
                                pt1 := pt1^.next;
                                continue := pt1<>nil
                               end
                              else
                               if pt1^.val=pt2^.val
                                then
                                 begin
                                  new(q);
                                  q^.val := pt1^.val;
                                  q^.next := qp;
                                  qp := q;
                                  pt1 := pt1^.next;
                                  pt2 := pt2^.next;
                                  continue := (pt1<>nil) and
                                                    (pt2<>nil)
                                 end
                        end;
                     if pt1=nil
                        then
                           pt1 := pt2;
                     while pt1<>nil do
                        begin
                           new(q);
                           q^.val := pt1^.val;
                           q^.next := qp;
                           qp := q;
                           pt1 := pt1^.next
                        end
                        end
                     end;  (* of OR *)
      'N','n':         (* NOT *)
               begin
                  while continue do
                     begin
                        if pt1^.val>pt2^.val
                           then
                              begin
                               pt2 := pt2^.next;
                               continue := pt2<>nil
                              end
                           else
                              if pt1^.val<pt2^.val
                               then
                                begin
                                 new(q);
                                 q^.val := pt1^.val;
                                 q^.next := qp;
                                 qp := q;
                                 pt1 := pt1^.next;
                                 continue := pt1<>nil
                                end
                               else
                                if pt1^.val=pt2^.val
                                 then
                                  begin
                                   pt1 := pt1^.next;
                                   pt2 := pt2^.next;
                                   continue := (pt1<>nil) and
                                                    (pt2<>nil)
                                  end
                     end;
                  while pt1<>nil do
                     begin
                        new(q);
                        q^.val := pt1^.val;
                        q^.next := qp;
                        qp := q;
                        pt1 := pt1^.next
                     end
               end  (* of NOT *)
   end;  (* of case *)
   ReWind(qp);
   p1 := qp
end;   (* of join *)

procedure OutList(ptr : link; var aa : integer);

var p : link;
begin
   p := ptr;
   aa := 0;
   writeln;
   while p<>nil do
     begin
        aa := aa+1;
        if aa mod 13 = 0
           then
              writeln(p^.val :5)
           else
              write(p^.val :5);
        p := p^.next
     end;
  writeln;
  writeln
end;  (* of OutList *)

procedure DictList(var where : text);
(* TO LIST DICTIONARY   *)

const NoOfLines = 64;
      WordsPerLine = 4; (* Change constants to suit page size *)
                        (* See also line 700 *)

type list = array[1..384] of word;

var      num,i : integer;
    OldEntry : dic;
    WordList : list;
begin
   reset(dict);
   rewrite(dlist);
   i := 0;
   while not eof(dict) do
      begin
         for num:=1 to NoOfLines*WordsPerLine do
            begin
               OldEntry := dict^;
               while (dict^.cont=true)and(not eof(dict)) do
                  get(dict);
               if not eof(dict)
                  then
```

```
                begin
                   WordList[num] := OldEntry.name;
                   get(dict)
                end
             else
                WordList[num] := '                    ';
          end;
       for num:=1 to NoOfLines do
          writeln(where,WordList[num],WordList[NoOfLines+num],
                  WordList[2*NoOfLines+num],
                                   WordList[3*NoOfLines+num]);
         (* Extent this list for more words per line *)
       i := i+NoOfLines*WordsPerLine
    end;
writeln;
write('Dictionary written to file.');
writeln(' To obtain a hard copy run "outdict".');
(* 'outdict' simply prints out the file 'dlist'. *)
   writeln
end;  (* of DictList  *)

procedure TwoCols (var F,G : text);

const rows = 8;
      TwiceRows = 16;
      cols = 40;

type   ChLink = ^chstack;
     chstack = record
                  ch : char;
                  next : ChLink
               end;
     lines = array[1..TwiceRows] of ChLink;

var    pt,here : ChLink;
    lin,StartLin : lines;
    LineNo,CharNo : integer;
    ch : char;

procedure reverse(var ptr : ChLink);

var p,q,pt : ChLink;
begin
   p := ptr;
   pt := nil;
   while p <> nil do
      begin
         new(q);
         q^.ch := p^.ch;
         q^.next := pt;
         pt := q;
         p := p^.next
      end;
   ptr := pt
end;  (* of reverse *)

begin
   reset(F);
   if not eof(F)
      then
         begin
            page(G);
            writeln;
            writeln('Output in two column ''Xerox'' label format.');
            writeln
         end;
   while not eof(F) do
      begin
         mark(here);
         for LineNo := 1 to 2*rows do
            begin
      StartLin[LineNo] := nil;
      if not eof(F) then
      while not eoln(F) do
         begin
            read(F,ch);
            new(lin[LineNo]);
            lin[LineNo]^.ch := ch;
            lin[LineNo]^.next := StartLin[LineNo];
            StartLin[LineNo] := lin[LineNo]
         end;
      if not eof(F)
         then
            readln(F);
      reverse(StartLin[LineNo]);
   end;

for LineNo := 1 to rows do
   begin
      CharNo := 0;
      pt := StartLin[LineNo];
      while (pt <> nil) and (CharNo < cols) do
         begin
            write(G,pt^.ch);
            pt := pt^.next;
            CharNo := CharNo + 1
         end;
      pt := StartLin[LineNo + rows];
      if pt <> nil
         then
            while CharNo < cols do
               begin
                  write(G,' ');
                  CharNo := CharNo +1
               end;
      while pt <> nil do
         begin
            ch := pt^.ch;
            write(G,ch);
            pt := pt^.next
         end;
      writeln(G)
   end;
release(here);
end
end; (* of TwoCols *)

procedure GetFromDict(var FirstWord,NumWords : integer);

var
    ch,action,option      : char;
    n,ChCount,PointerNum,NumberFound : integer;
    name,signame : word;
    AllCaps : boolean;
begin
writeln;
AllCaps := true;
ChCount := 0;
write('Enter word required or [.] ....   ');
repeat
   read(ch)
until ch<>' ';
if ch='\'
   then
      begin
         AllCaps := false;
         read(ch)
      end;
if ch='.'
   then
      begin    (*  "action"  *)
         while not eoln(input) do
            get(input);
         repeat
            writeln;
 writeln( 'Do you wish to LOOK UP the selected string,
           to RESTART the' );
            write('selection or to QUIT the dictionary ....  ');
            InlChar(action)
         until action in ['L','l','R','r','Q','q']
      end
   else
      begin    (* word *)
         action := 'W';
         repeat
            ChCount := ChCount + 1;
            if ChCount > 1
               then
                  read(ch);
            if AllCaps and (ch in ['a'..'z'])
               then
                  name[ChCount] := chr(ord(ch)-32)
               else
                  name[ChCount] := ch
         until eoln(input) or (ChCount = 20);
         if not eoln(input)
            then
               readln;
         for n:=ChCount+1 to 20 do
            name[n] := ' '
      end;
if action in ['L','l']
   then
      FirstWord := -1  (* look up *)
   else
      if action in ['R','r']
         then
            FirstWord := -2  (* restart *)
         else
           if action in ['Q','q']
         then
            FirstWord := 0  (* quit *)
         else
```

```
            if name='***
                (* special word *)
               then
                  begin
                     writeln;
                     writeln('***  ALL ITEMS  ***');
                     writeln;
                     repeat
                      write('Is this correct  [YES or NO] ....  ');
                      InlChar(option)
                     until option in ['Y','y','N','n'];
                     if option in ['Y','y']
                      then
                       FirstWord := HiTag
                      else
                       GetFromDict(FirstWord,NumWords)
                  end
               else
                  begin  (* a real word *)
                     reset(dict);
                     NumberFound := 0;
                     PointerNum := 0;
                     writeln;
                     signame := '                         ';
                     while (name >= signame) and not eof(dict) do
                      begin
                       if name=signame
                        then
                         begin
                          writeln(dict^.name);
                          NumberFound := NumberFound+1
                         end;
                       while (dict^.cont=true) do
                        get(dict);
                       if (PointerNum > 0) and not eof(dict)
                        then
                         get(dict);
                       PointerNum := PointerNum+1;
                       for n:=1 to ChCount do
                        signame[n] := dict^.name[n];
                       for n:=ChCount+1 to 20 do
                        signame[n] := ' ';
                      end;
                     writeln;
                     if NumberFound=0
                      then
                       begin
  writeln( 'Word not found in your dictionary; try again.' );
                        writeln;
                        GetFromDict(FirstWord,NumWords)
                       end
                             else
                              begin
                               repeat
                                if NumberFound = 1
                                 then
        write( 'Is this word correct  [YES or NO] ....  ' )
                                 else
        write( 'Are ALL these words required  [YES or NO] .... ' );
                                InlChar(option)
                               until option in ['Y','y','N','n'];
                               if option in ['Y','y']
                                then
                                 begin
                                  FirstWord := PointerNum -
                                               NumberFound;
                                  NumWords := NumberFound
                                 end
                                else
                                 GetFromDict(FirstWord,NumWords)
                              end
                       end
end;   (* of GetFromDict  *)

begin (* MAIN PROGRAM *)
   rewrite(scratch);
   rewrite(AddressFile);
   reset(bank);
   count := HiTag;
   LineNo := 0;
   AddLineNo := 0;
   FileAssigned := false;
   writeln;
   writeln('To retrieve ITEMS from the BIBLIOGRAPHY.');
(* TO SEARCH BY AUTHORS and KEYWORDS *)
   writeln;
   reset(dlist);
   if dlist^ = '~'
      then
         InlInt(dlist,TopItem)
      else
         begin
            TopItem := 0;
            writeln('Counting, please wait.');
            writeln;
            writeln;
            while not eof(bank) do
               begin
                  TopItem := TopItem +1;
                  get(bank)
               end;
            rewrite(dlist);
            writeln(dlist,'~',TopItem : 5);
            writeln(dlist);
            writeln(dlist);
     writeln(dlist,'Your DICTIONARY must first be compiled by running');
    writeln(dlist,' the HARD COPY option of ''bibout''.');
            writeln(dlist);
            writeln(dlist)
         end;
writeln('The BIBLIOGRAPHY currently holds ',TopItem,'  ITEMS.');
repeat
   writeln;
   10: repeat
 writeln( 'Do you wish to obtain a HARD COPY of the current dictionary,'
          write('to SEARCH for items or to FINISH ....  ');
          InlChar(MainOpt)
        until MainOpt in ['H','h','S','s','F','f'];
   writeln;
   if MainOpt in ['H','h']
      then
         begin
            DictList(dlist);
            MainOpt := 'F'
         end;
   if MainOpt in ['S','s']
      then
         begin
            repeat
               writeln;
               writeln('Do you wish to search by item NUMBER');
               write('or by use of the DICTIONARY ....  ');
               InlChar(NDOption)
            until NDOption in ['N','n','D','d'];
            writeln;
            repeat
               writeln;
               write('Output to TERMINAL or to scratch FILE ....  ');
               InlChar(device)
            until device in ['T','t','F','f','S','s'];
            writeln;
            if device in ['T','t']
               then
                  FileStyle := 'T';
            if (device in ['F','f','S','s']) and not FileAssigned
               then
                  repeat
                     writeln('Is the desired output');
                     write('an ITEM list,');
                     writeln('  ''the full item being given'' ');
                     write('a REFERENCE list,');
                     writeln('  ''only the reference part being given''
                     write('or an address list suitable');
                     write(' for ENVELOPE addressing .... ');
                     InlChar(FileStyle);
                     FileAssigned := true
                  until FileStyle in ['I','i','R','r','E','e'];
            if FileStyle in ['R','r']
               then
                  begin
                     writeln(scratch,'.hy 0'); (* NROFF commands *)
                     writeln(scratch,'.na');
                     writeln(scratch,'.sp 2');
                     writeln(scratch,'.de nr');
                     writeln(scratch,'.sp');
                     writeln(scratch,'.ne 6');
                     writeln(scratch,'.ti -5');
                     writeln(scratch,'..');
                     writeln(scratch,'.ne 10');
                     writeln(scratch,'\:References.\:');
                     writeln(scratch,'.sp 2');
                     writeln(scratch,'.in +5')
                  end;
            writeln;
            case NDOption of
               'D','d' : begin
  writeln('Words are looked up in ');
  writeln('the dictionary and a list of reference numbers' );
  writeln( 'containing the given word is shown on the terminal.' );
  writeln;
  write( 'The special "word", [***] will match with all the words' );
```

```
writeln(' in the dictionary.');
writeln;
write('Logical combination of ');
writeln( 'author and keywords continue until you wish' );
w   ln('to terminated the search.');
v    ln;
wr..eln( 'To terminate a search answer the prompt with a full
        stop [.].' );
writeln;
                          repeat
                           writeln;
                           writeln('New sequence.');
                           writeln;
                           NumSoFar := 0;
                           mark(here);
                           GetFromDict(NFromDict,NumW);
                           if NFromDict > 0  (* a real word *)
                            then
                             begin
                              GetDict(NFromDict,FirstLink);
                              if NumW > 1
                               then
                                repeat
                                 NFromDict := NFromDict + 1;
                                 GetDict(NFromDict,SecondLink);
                                 join(FirstLink,SecondLink,'O');
                                 NumW := NumW - 1
                                until NumW = 1;
                              OutList(FirstLink,NumSoFar);
                              while NFromDict > 0 do
                               begin
                                GetFromDict(NFromDict,NumW);
                                if NFromDict > 0 (* a real word *)
                                 then
                                 begin
                                  GetDict(NFromDict,SecondLink);
                                  if NumW > 1
                                   then
                                    repeat
                                     NFromDict := NFromDict + 1;
                                     GetDict(NFromDict,ThirdLink);
                                     join(SecondLink,ThirdLink,'O');
                                     NumW := NumW - 1
                                    until NumW = 1;
                                  OutList(SecondLink,NumSoFar);
                                  repeat
    write( 'AND, OR or NOT .....  ?            ' );
                                   InlChar(LogicAction)
                                  until LogicAction in ['A','a','O',
                                        'o','N','n'];
                                  join(FirstLink,SecondLink,
                                       LogicAction);
                                  OutList(FirstLink,NumSoFar)
                                 end
                               end;
                              if ((NumSoFar > 0) and (NFromDict = -1))
                               then  (* look up *)
                                begin
                                 writeln;
    writeln('Search in progress for',NumSoFar :8, ' Items');
                                 writeln;
                                 ptl := FirstLink;
                                 while ptl<>nil do
                                  begin
                                   GetRef(ptl^.val,FileStyle);
                                   ptl := ptl^.next
                                  end;
                                 if FileStyle in ['I','i','R','r',
                                        'E','e']
                                  then
                                   begin
                                    writeln;
     writeln( 'ITEMS written to SCRATCH FILE.' );
                                    writeln
                                   end;
                                  release(here)
                                 end;
                             end
                           until NFromDict=0  (* quit *)
                         end;
            'N','n' : begin  (* TO SEARCH BY NUMBER *)
writeln;
writeln('ITEMS may be called by number.');
w   ln('A whole block of ITEMS may be called;');
v    ('to do this answer this prompt with');
wr..eln(' minus one [-1].');
writeln;
writeln( 'To quit: answer prompt with a zero [0]. ' );
                          repeat
                           writeln;
      write('Number of ITEM to be referenced.....  ' );
                                        InlInt(input,n);
                                        writeln;
                                        if n = -1
                                         then
                                          begin
                                           writeln;
                                           writeln('To output a block of
                                                    ITEMS.');
      writeln( 'Give the LOW ITEM number ,then the HIGH number.' );
                                           write('LOW number  ....  ');
                                           InlInt(input,low);
                                           write('HIGH number ....  ');
                                           InlInt(input,high);
                                           if (low=0) or (high=0)
                                            then
                                             begin  (* an escape *)
                                              low := 1;
                                              high := 0;
                                              n := 0
                                             end;
                                           if low <= high
                                            then
                                             begin
                                              writeln;
                                              writeln('Search in progress');
                                              writeln;
                                              for n:=low to high do
                                               GetRef(n,FileStyle)
                                             end
                                          end
                                         else
                                          if n > 0
                                           then
                                            begin
                                             writeln;
                                             writeln('Search in progress.');
                                             writeln;
                                             GetRef(n,FileStyle)
                                            end
                                       until n=0
                                     end
                          end  (* of case MDOption *)
                        end
until MainOpt in ['F','f'];
if FileStyle in ['R','r']
   then
      begin
         writeln(scratch,'.in -5');
         writeln;
  writeln( 'The output file ''scratch'' contains the references and
            the' );
         writeln('instructions for the word processing program
                  ''nroff''.');
         writeln;
  writeln( 'An attempt has been made to reintroduce lower case
            letters.' );
         writeln('To obtain your output run   ''nroff scratch''  ');
         writeln;
    writeln( 'If all is not well edit scratch and run ''nroff
              scratch'' again.' )
         writeln;
         writeln('When all is correct get the hard copy output
                  by  ');
         writeln('running ''nroff scratch |lpr''. ');
         writeln
      end;
   if FileStyle in ['E','e']
      then
         TwoCols(AddressFile,scratch);
   writeln;
   writeln;
   writeln('FINISHED.');
   writeln
end.  (* of program Bibout.p *)
```

```
program Bibupdate(input,output,bank,dict,scratch,
                                dlist,PendingTray,TempBank);
(* A non-interactive program which moves the contents of
'PendingTray' to the bibliography. Clever systems run this program
at night.
TempBank is made external because it grows to be as large as bank.
Diagnostics are written to 'scratch'.
Written by Tony Heyes, Blind Mobility Research Unit,
Department of Psychology, The University,
Nottingham, U.K..   *)

const    LineLn = 70;
      RowLn = 20;
```

```
      heap = 200;
      HiTag = 10000;
      stack = 50;
      NonDate = -1066;

type     string = packed array [1..LineLn] of char;
     item = record
               authors,titlel,title2,
               placel,place2 : string;
               date       : integer;
               keyl,key2  : string
            end;
     word = packed array [1..20] of char;
     row = array [1..RowLn] of integer;
     TagItem = record
                  tag : integer;
                  entry : item
               end;
     point = ^CoreTagItem;
     CoreTagItem = record
                      TagEntry : TagItem;
                      next     : point
                   end;
     dic = record
              name      : word;
              numbers   : row;
              cont      : boolean
           end;
     link = ^dentry;
     dentry = record
                 dline  : dic;
                 next   : link
              end;

var    bank,TempBank,addition : file of item;
    LastOne : item;
    PendingTray,correction : file of TagItem;
    first,here,p,pt,newp : link;
    efirst,now,ept,e,enewp : point;
    dlist,scratch  : text;
    TempDict,dict : file of dic;
    GotFromCore,dlistOK,InitialBuild,continue,move,same : boolean;
      n,TopItem,m,corr,reps,add,OldTotal : integer;

  procedure FromCore;

  var p : link;
  begin
     writeln(scratch,'      Fromcore');
     rewrite(dict);
     GotFromCore := true;
     p := first;
     while p<>nil do
        begin
           dict^ := p^.dline;
           put(dict);
           p := p^.next
        end
  end;  (* of FromCore *)

  procedure build(entry : item;n : integer);
       (* TO BUILD THE DICTIONARY *)

  var         str : string;
      NewEntry,OldEntry : dic;
      l,let,line,i      : integer;
      same,space,AlreadyHad,WordFound,LastWord     : boolean;
  begin
     for line:=1 to 3 do
        begin
           case line of
              1: str := entry.authors;
              2: str := entry.keyl;
              3: str := entry.key2
           end;
           l := 0;
           let := 0;
           if not ((str[1]=' ')and(str[2]=' '))
              then
                 repeat  (* not empty line *)
                    let := let+1;
                    LastWord := (((str[let]=' ') and
                                                 (str[let+1]=' '))
                                 or (let=LineLn-1));
                    WordFound := ((str[let]=',') or LastWord);
                    if not WordFound
                       then
                          begin
                             l := l+1;
                             if (l=1) and (str[let]=' ') then
                                l := 0
                                else
                                begin
                                 if l<21 then
                                   NewEntry.name[l] := str[let]
                                end
                          end
           else
              begin
                 for i:=l+1 to 20 do
                    NewEntry.name[i] := ' ';
                    (* fill up with spaces *)
                 if InitialBuild
                    then
                     begin     (* first entry *)
                      NewEntry.numbers[1] := n;
                      for i:=2 to RowLn do
                       NewEntry.numbers[i] := 0;
                      NewEntry.cont := false;
                      new(p);
                      p^.dline := NewEntry;
                      p^.next := nil;
                      first := p;
                      l := 0;
                      InitialBuild := false
                     end
                    else
                     begin
                      OldEntry := first^.dline;
                      pt := first;
(* move pt past all words before the new entry *)
                      while (pt^.next<>nil) and
                      (NewEntry.name>=pt^.next^.dline.name) do
                       pt := pt^.next;
                      OldEntry := pt^.dline;
                      same := OldEntry.name=NewEntry.name;
                      space := OldEntry.numbers[RowLn]=0;
                      AlreadyHad := false;
                      if same then
                        begin
                          i := RowLn;
                          while OldEntry.numbers[i] = 0 do
                            i := i-1;
                          if OldEntry.numbers[i] = n then
                            AlreadyHad := true
                        end;
                      if not AlreadyHad then
                       begin (* if keyword has author name only
                       one dic|if (same and (not space))
                        then
                         begin
(* new entry already in dict but no space in the string *)
                           OldEntry.cont := true;
                           pt^.dline := OldEntry
                         end;
                        if same and space
                        then
                         begin
   (* new entry already in dict AND space in the number string *)
                           i := 0;
                           repeat
                                i := i+1
                              until OldEntry.numbers[i]=0;
                              OldEntry.numbers[i] := n;
                              pt^.dline := OldEntry
                             end
                           else
                            begin
  (* a new word for the dictionary OR a repeat of an old word *)
                              NewEntry.numbers[1] := n;
                              NewEntry.cont := false;
                              for i:=2 to RowLn do
                               NewEntry.numbers[i] := 0;
                              new(newp);
                              newp^.dline := NewEntry;
                              if NewEntry.name<first^.dline.name
                               then
                                begin   (* new head of the list *)
                                 newp^.next := first;
                                 first := newp;
                                end
                               else
                                begin   (* slot entry into list *)
                                 newp^.next := pt^.next;
                                 pt^.next := newp
                                end
                             end
                          end;  (* of AlreadyHad *)
                        l := 0
                       end
              end
     until LastWord
```

```
      end
end; (* of build *)

procedure merge;
     (* to merge dict in core with existing dict on file *)

var   continue : boolean;
    j,jj : integer;
    NewEntry : dic;
begin
   writeln(scratch,'      Merge');
   rewrite(TempDict);
   reset(dict);
     (* copy to scratch with additions *)
   pt := first;
   continue := (not eof(dict)) and (pt^.next<>nil);
   while continue do
      begin
         if dict^.name<pt^.dline.name
            then
               begin
                  TempDict^ := dict^;
                  put(TempDict);
                  get(dict);
                  continue := not eof(dict)
               end;
         if dict^.name>pt^.dline.name
            then
               begin
                  TempDict^ := pt^.dline;
                  put(TempDict);
                  pt := pt^.next;
                  continue := pt<>nil
               end;
         if dict^.name=pt^.dline.name
            then
               begin
                  dict^.cont := true;
                  TempDict^ := dict^;
                  put(TempDict);
                  get(dict);
                  continue := not eof(dict)
               end
      end;
    hile not eof(dict) do
      begin
         TempDict^ := dict^;
         put(TempDict);
         get(dict)
      end;
   while pt<>nil do
      begin
         TempDict^ := pt^.dline;
         put(TempDict);
         pt := pt^.next
      end;

   rewrite(dict);
   reset(TempDict);
     (* copy back to dict and squeeze *)
   while not eof(TempDict) do
      begin
         NewEntry := TempDict^;
         if (NewEntry.numbers[RowLn]>0) or (NewEntry.cont=false)
            then
               begin
                  dict^ := NewEntry;
                  put(dict);
                  get(TempDict)
               end
            else
               begin
                  get(TempDict);
                  if not eof(TempDict)
                     then
                        begin
                           for j:=2 to RowLn do
           if NewEntry.numbers[j]=0
            then
             begin
              NewEntry.numbers[j] := TempDict^.numbers[1];
              for jj:=1 to RowLn-1 do
               TempDict^.numbers[jj] := TempDict^.numbers[jj+1]
              TempDict^.numbers[RowLn] := 0
             end;
         if TempDict^.numbers[1]=0
            then
             begin
              NewEntry.cont := false;
              get(TempDict);
              dict^ := NewEntry;
                                                  put(dict)
                                                 end
                                                else
                                                 begin
                                                  dict^ := NewEntry;
                                                  put(dict)
                                                 end
                                          end
                                    end
                              end;
                     rewrite(TempDict)
end;    (* of merge *)

begin (* MAIN PROGRAM *)
   reset(PendingTray);
   reset(bank);
   dlistOK := false;
   rewrite(scratch);
   writeln(scratch);
   writeln(scratch,'No new additions.');
   writeln(scratch);
   GotFromCore := false;
   corr := 0;
   reps := 0;
   add := 0;
   TopItem := 0;
   reset(dlist);
   if dlist^ = '~' then dlistOK := true;
   if eof(PendingTray)
      then
         begin
          if not dlistOK then
            while not eof(bank) do
              begin
                TopItem := TopItem + 1;
                get(bank)
              end
         end
         else
         begin
(* divide PendingTray into corrections and additions *)
         rewrite(correction);
         rewrite(additions);
         rewrite(dict);
         rewrite(scratch);
         dlistOK := false;
         while not eof(PendingTray) do
            if PendingTray^.tag<HiTag
               then
                  begin
                     write(correction,PendingTray^);
                     corr := corr+1;
                     get(PendingTray)
                  end
               else
                  begin
                     write(addition,PendingTray^.entry);
                     add := add+1;
                     get(PendingTray)
                  end;
         reset(correction);
         writeln(scratch,'Corrections ',corr :5,'  Additions ',add:5);

         while not eof(correction) do
            begin
(* order correction into core in batches of 'stack'  *)
               writeln(scratch,'To deal with corrections');
               mark(now);
               n := 1;
               new(e);
               e^.TagEntry := correction^;
               e^.next := nil;
               efirst := e;
               get(correction);
               while (not eof(correction)) and (n<stack) do
                  begin
                     n := n+1;
                     new(enewp);
                     enewp^.TagEntry := correction^;
                     if correction^.tag<efirst^.TagEntry.tag
                        then
                           begin   (* new head of list *)
                            enewp^.next := efirst;
                            efirst := enewp
                           end
                        else
                           begin
(* move pointer ept to correct place, slot in new item *)
                            ept := efirst;
                            while (ept^.next<>nil) and
                            (correction^.tag>=ept^.next^.TagEntry.tag)
```

```
                    do
                    ept := ept^.next;
                   if correction^.tag=ept^.TagEntry.tag
                        then
                         ept^.TagEntry := correction^
(* replace with later correction, this is why items are sorted in
       this way *)
                       else
                        begin
                         enewp^.next := ept^.next;
                         ept^.next := enewp
                        end
                     end;
                  get(correction)
               end;  (*  n=stack or eof(correction)  *)
            write(scratch,'Corrections processed in ');
            writeln(scratch,'this batch  ',n :5);
(* first batch of items from 'correction' now in core and ordered  *)

  (*  now read bank to TempBank making changes from core.
    Items are labelled for later extraction by making the
          date = NonDate.
    Replacement items are passed to join additions.  *)
            write(scratch,'Copy bank to TempBank ....');
            rewrite(TempBank);
            reset(bank);
            OldTotal := 0;
            ept := efirst;
            while not eof(bank) do
               begin
                  OldTotal := OldTotal+1;
                  if (ept<>nil) and (ept^.TagEntry.tag=OldTotal)
                     then  (* we have found one to correct *)
                        begin
                         if ept^.TagEntry.entry.date<>NonDate
                          then  (* ie. it is not empty *)
                           begin
                (* Replacement item written to addition file *)
                            write(addition,ept^.TagEntry.entry);
                            reps := reps+1
                           end;
                        bank^.date := NonDate;
                        write(TempBank,bank^);
                        get(bank);
  (* Making the date = NonDate will remove the item when
    the last batch of corrections are processed  *)
                        ept := ept^.next;
                       end
                    else
                       begin
                        write(TempBank,bank^);
                        get(bank)
                       end
               end;
            release(now);
            writeln(scratch,' O.K.');

  (*  read TempBank back to bank  *)
            write(scratch,'Copy TempBank to bank ....');
          rewrite(bank);
          reset(TempBank);
          while not eof(TempBank) do
             if eof(correction) and (TempBank^.date=NonDate)
                then
                   get(TempBank)  (* removes corrected items *)
                else
                   begin
                      write(bank,TempBank^);
                      get(TempBank);
                   end;  (* of reading back to bank *)
          writeln(scratch,' O.K.');
          rewrite(TempBank)
       end;  (* return for more corrections  *)

     rewrite(correction);
     reset(addition);
     while not eof(addition) do
        begin
(* order additions alphabetically into core in batches of 'stack'  *)
          writeln(scratch,'To deal with additions.');
          if reps>0
             then
                writeln(scratch,'These include ',reps :5,
                      ' replacements.');
          mark(now);
          n := 1;
          new(e);
          e^.TagEntry.entry := addition^;
          e^.next := nil;
          efirst := e;
          get(addition);
          while not eof(addition) and (n<stack) do
             begin
                n := n+1;
                new(enewp);
                enewp^.TagEntry.entry := addition^;
                move := ((enewp^.TagEntry.entry.authors
                        > efirst^.TagEntry.entry.authors) or
                        ((enewp^.TagEntry.entry.authors
                        = efirst^.TagEntry.entry.authors) and
                        (enewp^.TagEntry.entry.date
                        > efirst^.TagEntry.entry.date)));
                if not move
                   then  (* new head of list *)
                      begin
                       enewp^.next := efirst;
                       efirst := enewp
                      end
                   else
                      begin
(* move pointer ept to correct place, slot in new item *)
                       ept := efirst;
                       while (ept^.next<>nil) and
                            ((addition^.authors
                            > ept^.next^.TagEntry.entry.authors) or
                            ((addition^.authors
                            = ept^.next^.TagEntry.entry.authors) and
                            (addition^.date
                            > ept^.next^.TagEntry.entry.date))) do
                        ept := ept^.next;
                       enewp^.next := ept^.next;
                       ept^.next := enewp
                      end;
                get(addition)
             end;  (* n=stack or eof(addition)  *)
          writeln(scratch,'Additions processed in this batch  ',n :5);

(* now read bank to TempBank making additions from core *)
          write(scratch,'Copy bank to TempBank ....');
          reset(bank);
          rewrite(TempBank);
          ept := efirst;
          continue := (not eof(bank)) and (ept<>nil);
          while continue do
           begin
              if ((bank^.authors < ept^.TagEntry.entry.autho    or
                 ((bank^.authors = ept^.TagEntry.entry.autho    and
                 (bank^.date     < ept^.TagEntry.entry.date)))
                 then
                    begin
                     write(TempBank,bank^);
                     get(bank);
                     continue := not eof(bank)
                    end
                 else
                    begin
                     write(TempBank,ept^.TagEntry.entry);
                     ept := ept^.next;
                     continue := ept<>nil
                    end
             end;  (* of the merging of the core and the file *)
          while not eof(bank) do
             begin
                write(TempBank,bank^);
                get(bank)
             end;
          while ept<>nil do
             begin
                write(TempBank,ept^.TagEntry.entry);
                ept := ept^.next
             end;
          LastOne := bank^;
(* assigned to give LastOne a starting value *)
          writeln(scratch,' O.K.');

       (* now copy back to bank *)
          write(scratch,'Copy TempBank to bank ....');
          reset(TempBank);
          rewrite(bank);
          release(now);
          while not eof(TempBank) do
             begin
                same := ((TempBank^.authors=LastOne.authors)
                        and (TempBank^.title1=LastOne.title1)
                        and (TempBank^.title2=LastOne.title2
                        and (TempBank^.date=LastOne.date));
                if not same
                   then
                     write(bank,TempBank^);  (* rejects duplicates *)
                LastOne := TempBank^;
                get(TempBank)
             end;
```

```
            writeln(scratch,' O.K.');
            rewrite(TempBank)
          end; (* return for more additions *)
      end; (* of dependence on PendingTray *)


  (* TO BUILD THE DICTIONARY  *)
reset(bank);
reset(dict);
rewrite(addition);
rewrite(PendingTray);
if eof(dict)
   then
      begin
         n := 0;
         InitialBuild := true;
         m := 0;
         mark(here);
         writeln(scratch,'To build dictionary');
         while not eof(bank)  do
            begin
               n := n+1;
               m := m+1;
               build(bank^,n);
               get(bank);
               if m=heap
                  then
                     begin
                        if not GotFromCore
                           then
                            FromCore
                           else
                            merge;
                        release(here);
                        mark(here);
                        InitialBuild := true;
                        m := 0
                     end
            end;

           if not GotFromCore
              then
                 FromCore
              else
                 merge;
           release(here);
         end;
   if n > 0 then TopItem := n;
   if not dlistOK then
     begin
       rewrite(dlist);
       writeln(dlist,'~ ',TopItem : 5);
       writeln(dlist);
       write(dlist,'DICTIONARY must be compiled by running ');
       writeln(dlist,'the HARD COPY option of ''bibout''.');
       writeln(dlist);
     end
end.  (* of program Bibupdate.p  *)
```

Pascal Standards: Progress Report

By Jim Miner
1982-06-10

ISO Standard

The technical work on the ISO standard is complete. The final six-month vote on the first Draft International Standard (DIS 7185) closes later this year; we expect the standard to be approved. DIS 7185 is identical to the new British Standard approved by the British Standards Institution (BSI) on 1981-09-15, together with a French translation produced by AFNOR and French Pascalers. The English version should be available from your national standards organization (e.g., AFNOR, ANSI, BSI, DIN), although when I called ANSI in June 1982 they had not yet received it from BSI. In any case, the document "BS 6192: 1982" (price: £18) can be ordered from British Standards Institution, 2 Part Street, London W1A 2BS, United Kingdom (Telephone 01-629-9000; Telex 266933). Or, you can wait for the next edition of *A Practical Introduction to Pascal* by Ian Wilson and Tony Addyman. Tony tells me that it is due out later this year and that it will include the BSI standard in a second section. The French standard, "NFZ 65.300" (price unknown), can be ordered from Association Francaise de Normalisation, Tour Europe, Cedex 7, 92080 Paris La Defense, France.

U.S. Standards

The public comment period on the draft proposed American National Standard (dpANS) is over, and X3J9 is preparing its responses to comments received. As expected, X3J9 refuses to include conformant array parameters in the first ANSI standard. Except for this, the draft appears to be very similar in content to the ISO draft.

The IEEE, on 1981-09-17, adopted an early version of the ANSI draft. This has several technical deficiencies and differences from the ANSI and ISO/BSI documents, and therefore will almost certainly change.

Validation Suite

Several persons have submitted comments on the new Validation Suite (version 3.0) to Arthur Sale and Brian Wichmann. A few errors in the Suite were uncovered, and Brian has issued the "Status Report" printed below. Also included are some useful (unpublished) appendices to the BSI standard that Brian sent along.

Future Standards

With the current round of standards achieving approval, several committees are deciding what extensions to include in subsequent standards. Both the French and the U.S. committees are quite interested in pursuing extensions. The (ISO) Working Group 4 attempted at its meeting in October 1981 to define procedures for international coordination, but no agreement has been reached. This is unfortunate because lack of coordination, but no agreement has been reached. This is unfortunate because lack of coordination now may mean that international standardization of extensions will be based more on political than technical considerations. This reminds me of Niklaus Wirth's characterization of formal standardization as "time-consuming and politics-infested."

I hope to print in future issues of *Pascal News* some of the extensions being considered by the U.S. committee (JPC). And I hope that members of other national bodies will do the same. Perhaps in this way you, the *users* of Pascal, will have a real chance to try out and comment on these proposals before they appear in a standard.

Status Report on Version 3.0 of the Pascal Test Suite

By B.A. Wichmann
82-4-1

The test suite was issued on the 8th January 1982. As a result, a large number of comments have been received. We hope to issue a new version in about four months: Arthur Sale is handling the additions while I am amending the existing tests.

Report on 3.0

At NPL, we keep a record of all agreed comments. Each one is classified (somewhat arbitrarily) as a Bug, Defect, Typo or Remark. The current count is 25 bugs (20 of which are of a less serious nature), 27 defects, 20 typos and 10 remarks. Please write in with your comments since it is an invaluable aid in improving the suite.

Rather than give a detailed report on each test for which improvements will be made, the list for which test output should be ignored is: 6.4.3.3-5, 6.5.1-1, 6.6.6.1-1, 6.6.6.5-1, 6.9.3.5.1-1 and (for safety) all of

section 6.6.3.7. Most of the other reports reflect inaccuracies in the comment describing the test or the possibility of one error masking another. Currently, complete testing is awkward due to the absence of compilers which adhere completely to the Standard.

Listing Errors

The listing sent to WG4 members and those that obtain the suite from NPL should note the following errors:

```
  Page no.   Test no.    line no.       error
   92        6.1.8-3      11          "A" missing in
                                        "DEVIATES"
  290        6.1.9-6      19      % should be commercial
                                        at sign
  297      6.7.2.3-4       1          "A" missing in
                                      "IMPLEMENTATION"
  299      6.8.2.3-2     5-16         lines missing,
                                    see below for text
Index       section 6.8.3.10 is misplaced.

The missing text on Page 299:
 5 {V3.0: New test. }
 6
 7 program t6p8p2p3d2(output);
 8 var
 9    string : packed array[1..3] of char;
10    i      : integer;
11 function sideeffect(c : char) : integer;
12 begin
13    string[i] := c;
14    i := i + 1;
15    sideeffect := i
16 end;
```

The new version

The major changes will be as follows:

a) removal of bugs/defects/typos;

b) use of local files rather than program parameters in 54 tests;

c) seven programs changed to remove the use of type real (which is unnecessary);

d) complete rewrite of the conformant array tests (which were to an earlier version of the standard);

e) alteration of initial comments to permit their use in a package for automatic production of a test report;

f) the inclusion of "pretests" for each error handling test to guard against the failure of an error handling test for the wrong reason;

g) additional tests where these are necessary, especially non-text file handling.

The Standard

This was published by BSI in February. It can be obtained from national standards bodies, and BSI can quote special rates for orders in quantity.

Two Appendices

Enclosed is a copy of two appendices which can usefully be added to the Standard. Also enclosed is a list of section numbers from which each error listed in Appendix D originates.

Index to errors listed in Appendix D

Error number 1 — Section 6.5.3.2
Error number 2 — Section 6.5.3.3
Error number 3 — Section 6.5.4
Error number 4 — Section 6.5.4
Error number 5 — Section 6.5.4
Error number 6 — Section 6.5.5
Error number 7 — Section 6.6.3.2
Error number 8 — Section 6.6.3.2
Error number 9 — Section 6.6.5.2
Error number 10 — Section 6.6.5.2
Error number 11 — Section 6.6.5.2
Error number 12 — Section 6.6.5.2
Error number 13 — Section 6.6.5.2
Error number 14 — Section 6.6.5.2
Error number 15 — Section 6.6.5.2
Error number 16 — Section 6.6.5.2
Error number 17 — Section 6.6.5.2
Error number 18 — Section 6.6.5.2
Error number 19 — Section 6.6.5.3
Error number 20 — Section 6.6.5.3
Error number 21 — Section 6.6.5.3
Error number 22 — Section 6.6.5.3
Error number 23 — Section 6.6.5.3
Error number 24 — Section 6.6.5.3
Error number 25 — Section 6.6.5.3
Error number 26 — Section 6.6.5.4
Error number 27 — Section 6.6.5.4
Error number 28 — Section 6.6.5.4
Error number 29 — Section 6.6.5.4
Error number 30 — Section 6.6.5.4
Error number 31 — Section 6.6.5.4
Error number 32 — Section 6.6.6.2
Error number 33 — Section 6.6.6.2
Error number 34 — Section 6.6.6.2
Error number 35 — Section 6.6.6.3
Error number 36 — Section 6.6.6.3
Error number 37 — Section 6.6.6.4
Error number 38 — Section 6.6.6.4
Error number 39 — Section 6.6.6.4
Error number 40 — Section 6.6.6.5
Error number 41 — Section 6.6.6.5
Error number 42 — Section 6.6.6.5
Error number 43 — Section 6.7.1
Error number 44 — Section 6.7.2.2
Error number 45 — Section 6.7.2.2
Error number 46 — Section 6.7.2.2
Error number 47 — Section 6.7.2.2
Error number 48 — Section 6.7.3
Error number 49 — Section 6.4.6
Error number 50 — Section 6.4.6
Error number 51 — Section 6.8.3.5
Error number 52 — Section 6.8.3.9
Error number 53 — Section 6.8.3.9
Error number 54 — Section 6.9.1
Error number 55 — Section 6.9.1
Error number 56 — Section 6.9.1
Error number 57 — Section 6.9.1
Error number 58 — Section 6.9.3.1
Error number 59 — Section 6.6.3.8

Appendix E to BS 6192

The BSI/ISO standard for Pascal includes an appendix D listing all the errors mentioned in the main document. This is very convenient for our work on compiler validation since a complying processor is required to state what action is taken for each of these errors. Therefore, we are producing a similar appendix for the implementation-defined features of Pascal. Appendix D does not indicate the section in the main text for each error listed. This is indicated in this appendix by means of section numbers in comment brackets.

Implementation-Defined

E.0 A complying processor is required to provide a definition of all the implementation-defined features of the language. To facilitate the production of this definition, all the implementation-defined aspects in clause 6 are described again in this appendix {5.1 (d)}

E.1 The value of each char-type corresponding to each allowed string-character. {6.1.7}

E.2 The subset of the real numbers denoted as specified by signed-real. {6.4.2.2 (b)}

E.3 The values of char-type. {6.4.2.2 (d)}

E.4 The ordinal numbers of each value of char-type. {6.4.2.2 (d)}

E.5 The point at which the file operations rewrite, put, reset and get are performed. {6.6.5.2}

E.6 The value of maxint. {6.7.2.2}

E.7 The accuracy of the approximation of the real operations and functions to the mathematical result. {6.7.2.2}

E.8 The default value of TotalWidth for integer type. {6.9.3.1}

E.9 The default value of TotalWidth for real-type. {6.9.3.1}

E.10 The default value of TotalWidth for Boolean-type. {6.9.3.1}

E.11 The value of ExpDigits. {6.9.3.4.1}

E.12 The value of the exponent character ('e' or 'E'). {6.9.3.4.1}

E.13 The case of each character of 'True' and 'False' for output. {6.9.3.5}

E.14 The effect of the procedure page. {6.9.5}

E.15 The binding of a file-type program parameter. {6.10}

Appendix F to BS 6192

The BSI/ISO standard for Pascal includes an appendix D listing all the erros mentioned in the main document. This is very convenient for our work on compiler validation since a complying processor is required to state what action is taken for each of these errors. Therefore, we are producing a similar appendix for the implementation-dependent features of Pascal. Appendix D does not indicate the section in the main text for each error listed. This is indicated in this appendix by means of section numbers in comment brackets.

Implementation-Dependent

F.0 A complying processor is required to provide documentation concerning the implementation-dependent features of the language. To facilitate the production of such documentation, all the implementation-dependent aspects specified in clause 6 are described again in this appendix. {5.1 (i and f)}

F.1 The order of evaluation of the index-expressions of an indexed variable. {6.5.3.2}

F.2 The order of evaluation of expressions of a member-designator. {6.7.1}

F.3 The order of evaluation of the member-designators of a set constructor. {6.7.1}

F.4 The order of evaluation of the operands of a dyadic operator. {6.7.2.1}

F.5 The order of evaluation, accessing and binding of the actual parameters of a function-designator. {6.7.3}

F.6 The order of accessing the variable and evaluating the expression of an assignment statement. {6.8.2.2}

F.7 The order of evaluation, accessing and binding of the actual-parameters of a procedure-statement. {6.8.2.3}

F.8 The effect of inspecting a textfile to which the page procedure was applied during generation. {6.9.5}

F.9 The binding of the variables denoted by the program parameters to entities external to the program. {6.10}

DISTRIBUTION OF THE EDISON SYSTEM

The Edison system is a portable software system that supports the development of programs written in the programming language Edison — a Pascal-like language that supports program modularity and concurrent execution on micoprocessors.

The Edison system includes an operating system, an Edison compiler, a screen editor, a text formatter, a print program, and a PDP 11 assembler written in the Edison language.

These programs can all be edited and recompiled on a PDP 11/23 microcomputer with 28 K words of memory, a VT 100 terminal, and a dual drive for 8-inch floppy disks.

The Edison compiler generates portable code which is interpreted by an assembly language kernel of 1800 words. The software can be moved to other 16-bit microcomputers with similar peripherals by rewriting the kernel.

The software is simple enough to be studied in detail at all levels of programming. It is described in a book entitled "Programming a Personal Computer" which includes the Edison language report and the program text of the kernel, the operating system, and the compiler.

For more information on the availability of the Edison system and the book, please write to:

Per Brinch Hansen
University of Southern California
Computer Science Department
University Park
Los Angeles, Calif. 90007

PASCAL CHOSEN AS SIL

The Languages group has completed its study and has recommended PASCAL as the Cray Research Systems Implementation Language (SIL). The SIL will be used to code the new Cray FORTRAN compiler and will likely be used for other Cray software products. The study investigated a number of languages, including PASCAL, C, FORTRAN (both CFT and CIVIC), and some other user-developed languages.

The first part of the study, completed in August, was a paper stating general requirements and assessing the suitability of the candidate languages. The choices were narrowed to two — PASCAL and C. FORTRAN did not have features considered necessary for systems programming. The other user-developed languages would encounter programmer resistance and raised maintenance concerns.

The second part of the study, recently completed, compared C and PASCAL. Code was written in both and compiled with existing compilers. PASCAL was chosen over C for three main reasons:

- Code generation: Existing and planned PASCAL compilers were designed for the CRAY-1; C was not. Major changes would be required to improve the C code generator to an acceptable level.
- Availability: PASCAL is available now on the AMDAHL and the CRAY-1 and a preliminary version of the University of Manchester compiler should be ready by the middle of 1982. C is running at Bell Labs, but licensing and cost have not been worked out.
- Language Proliferation: Most sites use (or would use) PASCAL; C does not have this demand. There is considerable field resistance to forcing sites to bring up another language processor merely to assemble the system.

If you have questions or are interested in copies of the SIL reports, please call Field Liaison/Support.

PASCAL: A PROBLEM SOLVING APPROACH

The computer language PASCAL was invented in 1971 by Niklaus Wirth to teach programming as a systematic discipline. Although not quite as widespread as the BASIC language, PASCAL has been adapted for use on most computers. PASCAL is easy to learn and use to detect common programming mistakes.

The new version of PASCAL, UCSD PASCAL™, has been developed for the Apple, TRS-80, LSI-11, and other microcomputers. Its power and ease of use have made it one of the most popular languages among the computer *cognoscenti*. Now this important computer language is available to all programmers in a new book by Elliot B. Koffman, PASCAL: A PROBLEM SOLVING APPROACH (Addison-Wesley; $14.95 trade paperback). Unlike most computer programming books, which simply teach the programming syntax, PASCAL: A PROBLEM SOLVING APPROACH emphasizes the structured, step-by-step design of computer programs. Both beginning programmers and those experienced in other languages such as BASIC will learn good programming techniques, good problem solving skills, the principles of "GOTO-free" or structured programming, and UCSD PASCAL.

An extensive market research survey conducted by Addison-Wesley has shown that UCSD PASCAL is quickly becoming one of the most popular microcomputer languages. With this book any programmer can learn to make full use of this powerful language. Some important features of PASCAL: A PROBELM SOLVING APPROACH by Elliot B. Koffman include:

- Appeals to both beginning and experienced programmers
- Stresses business oriented programs
- Complete coverage of all features of standard and UCSD PASCAL, including arrays, strings, sets, sequential and random access files
- Program style displays discuss important issues of programming style
- Display boxes summarize the syntactic form of each new language feature introduced
- Self-check exercises with selected answers integrated with the text
- Chapter summaries and discussions of common programming errors
- Additional programming problems at the end of each chapter
- Appendices include: the differences between standard and UCSD PASCAL, special identifiers and operators of PASCAL, using UCSD PASCAL, snytax diagrams of PASCAL statements

ABOUT THE AUTOR

Elliot B. Koffman is a Professor of Computer and Information Sciences at Temple University, Philadelphia. Dr. Koffman has organized and taught numerous seminars on computer language education across the nation. Dr. Koffman received his Bachelor's and Master's degrees from the Massachusetts Institute of Technology and earned his Ph.D. from Case Institute of Technology in 1967.

With Dr. Frank L. Friedman, Associate Professor of Computer and Information Sciences at Temple University, Dr. Koffman has co-authored three other computer language books: PROBLEM SOLVING AND STRUCTURED PROGRAMMING IN FORTRAN (Addision-Wesley, 1979), PROBLEM SOLVING AND STRUCTURED PROGRAMMING IN BASIC (Addison-Wesley, 1979), and PROBLEM SOLVING AND STRUCTURED PROGRAMMING IN PASCAL (Addison-Wesley, 1980). These three titles have sold over 300,000 copies combined.

MODULA — 2

Modula-2 (M2) (like Pascal and MODULA) was developed at the ETH-Zurich under direction of Niklaus Wirth (Institut fur Informatik).

1980: Fieldtest of M2
1981: Release of M2
1981: Production use of M2

M2 is a general purpose (system) programming language.

- Structured, modular, portable, readable, efficient, machine independent, flexible language.

FEATURES:

Modern syntax,
Module-structure,
Separate compilation,
Full type checking at compiling time,
Automatic version control and compatibility check,
Recursion,
Signed and unsigned integers,
Dynamic arrays (strings),
Procedures as variables or types,
Real-time (interrupts, DMA, processes, priorities, signals . . .)
Type transfer functions,
CPU and device register access,
Direct operating system calls,
Optimized machine code generation,
Multiprogramming: low-level facilities for specification of quasi-concurrent processes,
Dynamic creation of processes,
Fast interprocess communication and control with or without scheduler interactions,
Overlays,
Interactive linker,
Stand-alone programs,
Cross-reference generator,
Run time tests: array index bound, case index test, stack, heap,
Source code level debugger (procedure trace, process window, data window, source text window, core window),
all features of Pascal (block structure, type concept), except those that can be expressed in M2 itself,
and more.

Unsupported Features:

Gotos/labels,
Device or file input/output,
(Input/output) data conversion,
Heap (storage) — management,
Any process scheduling concept,
Mathematical/trisonometric functions,
Sets greater than the specific wordsize.

Currently supported (micro)processors:

- All PDP-11/LSI-11 with or without EIS/FIS/FPU/MMU options
- M 68000
- M 6809
- Lilith
- All other processors with advanced architecture via the portable M-Code compiler

Currently supported operating systems:

- RT11SJ, RT11FB (V4.0, V3B)
- Unix

Release of Modula-2 compilers

The design of the programming language Modula-2, started in 1977, was followed by implementation efforts of various compilers. A version designed for PDP-11 computer and its RT11 operating system was released in summer 1980. The following compilers were released on April 1, 1981. They will be distributed under licensing agreement with the purpose of protecting the language from arbitrary changes and extensions.

Computer	Operating System	Code	Compiler source
PDP(LSI)-11	RT-11	PDP-11	Modula-2
PDP(LSI)-11	UNIX	PDP-11	Modula-2
CDC-Cyber		PDP-11	Pascal6000
CDC-Cyber		M6809	Pascal6000
CDC-Cyber		M68000	Pascal6000
- Any -	- Any -	M-Code	Modula-2

The compilers were designed at the Institut fur Informatik of ETH, the Computation Center of ETH and the Australian Atomic Energy Commission. The compilers are distributed on Mag-Tape only.

The fee for each compiler is Sfr. 350 (A$ 150 for M2UNIX). The intention in distributing compilers for Modula-2 is to provide a modern tool for programming and thereby to advance the state of software engineering. The above fee must not be regarded as a price for the compiler, but rather as a handling charge and coverage of documentation, tape, package and postage.

If you wish to receive a compiler, request a license agreement:

Institut fur Informatik
ETH-Zentrum
CH-8092 Zurick

or for the UNIX implementation:

Departement of Computer Science attn. Dr. J. Tobias
University of New South Wales P. O. Box 1
Kensington, N.S.W. 2033 Australia

WRITENUM — A routine to output real numbers

By Doug Grover & Ned Freed
Harvey Mudd College
Claremont, Calif. 91711

Comments: Enclosed please find two articles for publication in P.N. The first, WRITENUM, is a new article. The second, TREEPRINT, is a re-draft of an article submitted last year. One change and one bug fix have been made.

A common problem encountered when writing utilities in standard Pascal is output of real numbers. Frequently it is desireable to output reals to a string and not to a file. Some Pascal implementations allow I/O to strings but this is a nonstandard feature. In addition, the standard output produced by WRITE and WRITELN is not flexible enough for many user applications.

For good readability, it is not enough to simply print out a number in scientific notation. First of all, it should be possible to constrain the modulus of the exponent to multiples of an integer. For example, the many calculators implement a mode called "engineering" notation where all exponents must be multiples of three. This is handy when converting output to SI units. For additional flexibility, we require that any multiple may be selected. An additional feature of modulus exponents is that for a given modulus N, up to N digits will appear in front of the decimal point. This makes values very readable when they appear in tabular form (more so than when they just appear in the correct columns).

Second, certain numbers should be handled in a special fashion. Numbers in the range [0.1, 1.0) should appear in the form 0.xxxxxx, instead of obeying the rules for modular exponents. This increases readability greatly. Numbers with more zeros after the decimal point should be converted to exponential form — it becomes difficult for the reader to keep track of where the point is.

Third, if a real can be printed as an integer, exponential format should be avoided. The point where integer format stops and exponential format takes over should be an adjustable parameter, as it may differ from application to application.

Finally, a facility for rounding numbers to a required number of digits should be provided. This facility should even extend to 0 digit accuracy, where only the magnitude is printed. The determination of whether a number is an integer should be based on the rounded result.

It is our belief that the usual concerns of columnar output are secondary to these considerations. If exact columns are necessary, they can be added as an afterthought. Left and right justification are trivial once the number is converted to a string of characters.

It is difficult to write a routine to satisfy these criteria in standard Pascal. A standard method of initially shifting and rounding the number to the desired accuracy fails because the Pascal ROUND function returns an integer which can cause machine overflow. While this can be overcome by initially truncating leading digits, the entire process of rounding introduces errors in the least significant bits of the mantissa. It is necessary to first convert the number into a string of digits, and then round the string. The string is then further manipulated to remove leading and trailing zeroes and place the decimal point in the proper position.

Another problem occurs with the initial determination of the exponent. Using the integer part of the common logarithm can fail since most machine implementations of LN are not accurate enough. The problem occurs when a number, neglecting its exponent, is close to 10 (e.g. 9999999.0). This can cause the number to be normalized to the wrong range.

We present a routine, WRITENUM, which we believe solves all these problems. It is vaguely based on a similar routine presented in Jensen and Wirth. It is entirely written in standard Pascal, with only two implementation dependent constants, the maximum number of digits allowed in a real and the natural log of 10. The log can be coded as an operation and the maximum number of digits can be set to 100 or so if portability is a problem.

WRITENUM uses an external function (STRINGADD) to perform its output. This method can be changed to straightforward string operations or even direct output statements via WRITE. In the listing, WRITENUM is imbedded in a small test program designed to demonstrate its features.

```
program test (input,output);

(*
A driver and test routine for the WRITENUM real number output routine
*)

type
  realtype = real;

var
  a : realtype; (* sample number *)
  b : integer;  (* desired digits of accuracy *)
  c : integer;  (* desired exponent modulus *)
  d : integer;  (* desired digits in integers *)

procedure stringadd (* sample routine to handle output of WRITENUM *)
  (x : char);

begin write (x); end;

(*
WRITENUM — a routine to output real numbers in
exponential format.

Written by Doug Grover and Ned Freed, 1-Feb-82

This routine takes a real number and outputs it in exponential format as a stream of ASCII characters. The
number of significant digits may be set as desired and
WRITENUM will round the output to that degree of
accuracy. In addition, the modulus of the exponent may
be set. This means that the exponent field may be constrained to be an integer multiple of any number. So-
called "engineering" format (modulus 3) and "scien-
```

tific" format (modulus 1) may both be generated. The exponent field is output only for non-zero exponents. Determination if a number will fit into a integer field occurs after the number is rounded. For the special case of a number in the range [0.1, 1.0), the numbers are output in the more natural 0.xxxxxx format. For the special case of an integer mantissa and a non-zero exponent, the numbers are output in the more natural xxxxxx.0Exxx format.

Note: 1) The value for the modulus can force up to that value of digits in front of the decimal point.

2) The precision of the output has a upper limit on the number of digits that can be output. This limit is selectable as a constant in the procedure.

```
    *)
procedure writenum (* outputs a number *)
  (x : realtype;   (*   the number to process for output *)
   n : integer;    (*   the number of digits of accuracy *)
   r : integer;    (*   the modulus of the exponent *)
   i : integer);   (*   the number of digits for integer output *)

const
  digitmax =   20;              (* maximum # of allowed digits *)
  zero     =  0.0;              (* zero *)
  one      =  1.0;              (* one *)
  ten      = 10.0;              (* ten *)
  lnoften  = 2.302582092994046; (* natural log of ten *)

var
  p  : integer; (* decimal point position *)
  e  : integer; (* exponent *)
  x1 : integer; (* counting variable *)

  digit : array [0..digitmax] of integer; (* digits in number *)

function tenof (* compute 10 raised to a power *)
  (e : integer) : realtype;

var
  i : integer;  (* bit counter *)
  t : realtype; (* result *)

begin  (* tenof *)
i := 0;
t := 1;
while e > 63 do begin
  t := t * 1.0E32;
  e := e - 32;
  end;
repeat
  if odd(e) then case i of
    0 : t := t * 1.0E1;
    1 : t := t * 1.0E2;
    2 : t := t * 1.0E4;
    3 : t := t * 1.0E8;
    4 : t := t * 1.0E16;
    5 : t := t * 1.0E32;
    end;
  e := e div 2;
  i := succ (i);
until e = 0;
tenof := t;
end;  (* tenof *)

begin  (* writenum *)

(* fix n, r, and i to be in range *)
if n > digitmax then n := digitmax;
if r > digitmax then r := digitmax;
if i > digitmax then i := digitmax;

(* special case x = 0 *)
if x = zero then stringadd ('0')
else begin
  if x < zero then begin
    stringadd ('-');
    x := -x;
    end;

  (* determine exponent and adjust number *)
  e := trunc (ln (x) / lnoften);
  if e > 0 then x := x / tenof (e)
  else x := x * tenof (-e);
  if x < one then begin
    e := pred (e);
    x := x * ten;
    end;

  (* put digits in array *)
  digit[0] := 0;
  x1 := 1;
  while x1 < n do begin
    digit[x1] := trunc (x);
    x := (x - digit[x1]) * ten;
    x1 := succ (x1);
    end;

  (* round the array representation of number *)
  digit[n] := round (x);
  while (n > 0) and (digit[n] = 10) do begin
    n := pred (n);
    digit[n] := succ (digit[n]);
    end;

  (* eliminate trailing zeroes *)
  while (n > 0) and (digit[n] = 0) do n := pred (n);

  (* handle rounding of all digits (all 9's) *)
  if n = 0 then begin
    n := 1;
    digit[n] := 1;
    e := succ (e);
    end;

  (* determine if integer output is possible *)
  if (n <= succ (e)) and (e < i) then begin
    p := succ (e);
    e := 0;
    end

  else begin

    (* allow 0.x format - a special case *)
    if e = -1 then begin
      e := 0;
      p := 0;
      stringadd ('0');
      stringadd ('.');
      end

    (* fix decimal point for modulus exponent *)
    else begin
      p := e mod r;
      if (e < 0) and (p <> 0) then p := r + p;
      e := e - p;
      p := succ (p);
      end;
    end;

  (* output digits from array *)
  x1 := 1;
  while n < p do begin
    n := succ (n);
    digit[n] := 0;
    end;
  repeat
    stringadd ( chr (digit[x1] + ord('0')));
    x1 := succ (x1);
  until x1 > p;
  if p < n then begin
    stringadd ('.');
    repeat
      stringadd ( chr (digit[x1] + ord('0')));
      x1 := succ (x1);
      until x1 > n;
    end;

  (* output exponent *)
  if e <> 0 then begin
    (* output optional '.0' for integer mantissa *)
    if p >= n then begin
      stringadd ('.');
      stringadd ('0');
      end;
    (* output exponent flag and sign *)
    stringadd ('E');
    if e < 0 then begin
      stringadd('-');
      e := -e;
      end;

    (* put exponent into array *)
    x1 := 0;
    repeat
      x1 := succ (x1);
      digit[x1] := e mod 10;
      e := e div 10;
    until e = 0;

    (* output exponent magnitude from array *)
    repeat
      stringadd ( chr (digit[x1] + ord ('0')));
      x1 := pred (x1);
    until x1 = 0;
    end;
  end;
end;  (* writenum *)

begin  (* test *)
repeat
  writeln
    ('Input #, accurate digits, exponent modulus, integer digits
  readln (a,b,c,d);
  write ('Number is - [');
  writenum (a,b,c,d);
  writeln (']');
until false;
end.  (* test *)
```

TREEPRINT — A Package to Print Trees on any Character Printer

By Ned Freed & 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.

Recursive structure are handled in TREEPRINT by checking each node with its ancestors. If it appears somewhere else in the diagram, no lower nodes are printed. This prevents fatal infinite loops that might otherwise occur.

The only problem in TREEPRINT at present 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 pic-
    torial 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 ap-
    peared 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.

Revision history:

1-Sep-81 : Fixed a problem in the output step that caused simple structures printed upside down to lose some connections. /nf

1-Dec-81 : Added code to check for recursive references and stop Position so that Treeprint does not hang in an infinite loop. /nf

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 environment 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 similar 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. 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 to 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, slines, 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 checkref  (N : nodeptr; currentref : reflink)
                       : boolean;

        (* Checkref is a function which checks whether a pointer
           into the user's data structure has already been used
           somewhere else in the tree. If so, Position should not
           examine lower nodes for this pointer -- they have
           already been taken care of elsewhere. This will prevent
           certain types of recursive references from getting
           Position into an infinite loop. This does, however,
           require a lot of time. *)

    var
        H : refhead;
        L : reflink;
        goon : boolean;

    begin  (* Checkref *)
        H := lines;
        goon := true;
        while (H <> nil) and goon do
        begin
            L := H^.first;
            while (L <> nil) and goon do
            begin
                if (L <> currentref) and (L^.pnode = N) then
                    goon := false;
                L := L^.next;
            end;
            H := H^.next;
        end;
        checkref := goon;
    end; (* Checkref *)

    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;
            if checkref (N,L) then
            begin
                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 = N;
            end;
            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);
                            if needright then
                            begin
                                right := left;
                                needright := false;
                            end
                            else left^.ustem := true;
                            over := over + 2;
                        end;
                    until (over > lastover) or (nodecount <= 0);
                    pos := (left^.pos + right^.pos) div 2;
                end
                else
                    L^.lstem := false;
                if pos > maxposition then maxposition := pos
                else
                    if pos < minposition then minposition := pos;
                L^.pos := pos;
            end;  (* if N = nil *)
        end;  (* Position *)

begin  (* Treeprint *)

    (* Initialize various variables. *)

    lines := nil;
    minposition := 0;
    maxposition := 0;
    charp := 0;

    (* Do various width and length calculations. *)

    if pagesize > 512 then pagesize := 512;
    width := abs (horikeylength) + 4;
    stemlength := abs (vertkeylength) + 1;
    vertnodelength := 3 * abs (vertkeylength) + 4;
    if (width mod 2) = 0 then width := width + 1;
    pagewidth := pagesize div width;

    (* Construct our data structure and compute positions. *)

    oldL := position (tree,lines,0);

    (* If the horizontal reverse option is selected, reverse
```

```
    every node on every line of the data structure. It is
    also necessary to switch around the states of the USTEM
    flags that tell who connects above a given node. *)

  if horikeylength < 0 then
  begin
      H := lines;
      while H <> nil do
      begin
          H^.first^.pos := maxposition -
                           H^.first^.pos + minposition;
          if H^.first^.ustem then
          begin
              H^.first^.ustem := false;
              endloop := true;
          end
          else
              endloop := false;
          L := nil;
          while H^.first^.next <> nil do
          begin
              H^.first^.next^.pos := maxposition -
                  H^.first^.next^.pos + minposition;
              if H^.first^.next^.ustem then
              begin
                  if not endloop then
                  begin
                      H^.first^.next^.ustem := false;
                      endloop := true;
                  end;
              end
              else
                  if endloop then
                  begin
                      H^.first^.next^.ustem := true;
                      endloop := false;
                  end;
              oldL := H^.first^.next;
              H^.first^.next := L;
              L := H^.first;
              H^.first := oldL;
          end;
          H^.first^.next := L;
          H := H^.next;
      end;
  end;

  (* If the vertical reverse option is selected, reverse the
     entire tree on the vertical axis by flipping all the
     head nodes along the edge. Arc reversal is handled in
     the actual arc generation routines. They will scan the
     previous line of info instead of the current one. *)

  slines := lines;
  if vertkeylength < 0 then
  begin
      H := nil;
    while lines^.next <> nil do
    begin
        D := lines^.next;
        lines^.next := H;
        H := lines;
        lines := D;
    end;
    lines^.next := H;
end;

(* Break up entire width into pages and loop over each. *)

startposition := minposition;
while startposition <= maxposition do
begin
    page (treefile);
    H := lines;
    while H <> nil do
    begin  (* Loop over all lines possible. *)
        oldL := H^.first;
        repeat  (* Find a node on current strip. *)
            endloop := true;
            if oldL <> nil then
                if oldL^.pos < startposition then
                    begin  (* Reject this node. *)
                        oldL := oldL^.next;
                        endloop := false;
                    end;
        until endloop;
        for i := 1 to vertnodelength do
        begin  (* Loop for each print line in a node. *)
            L := oldL;
            p := startposition;
            while (p < startposition + pagewidth) and
                (L <> nil) do
            begin  (* Scan for nodes we need to draw. *)
                if L^.pos = p then
                begin  (* Found node at current position. *)
                    if (i <= stemlength) then
                    begin  (* Draw upper stem part of node. *)
                        for w := 1 to (width div 2) do
                            cout (' ');
                        if ((vertkeylength < 0) and L^.lstem)
                            or ((vertkeylength >= 0) and
                            (H <> slines)) then cout ('*')
                        else cout (' ');
                        for w := 1 to (width div 2) do
                            cout (' ');
                    end
                    else
                    if (vertnodelength - i) < stemlength then
                    begin  (* Draw lower stem part of node. *)
                        for w := 1 to (width div 2) do
                            cout (' ');
                        if ((vertkeylength >= 0) and L^.lstem)
                           or ((vertkeylength < 0) and
                           (H <> slines)) then cout ('*')
                       else cout (' ');
                       for w := 1 to (width div 2) do
                           cout (' ');
                    end
                    else
                    if (i >= stemlength + 2)
                       and (i <= stemlength * 2) then
                        begin  (* Print node identifier. *)
                            cout ('*');
                            cout (' ');
                            cdump;
                            printkey (i - stemlength - 1,
                               abs (horikeylength), L^.pnode);
                            cout (' ');
                            cout ('*');
                        end
                    else
                        for w := 1 to width do cout ('*');
                    L := L^.next;
                end
                else
                    for w := 1 to width do cout (' ');
                p := p + 1;
            end;
            ctrim;
    end; (* for *)

    (* Select the proper line to obtain arc info from. *)

    if vertkeylength >= 0 then
    begin
        if H^.next <> nil
        then L := H^.next^.first
        else L := nil;
    end
    else L := H^.first;

    p := startposition;
    while (p < startposition + pagewidth) and (L <> nil) do
    begin
        endposition := L^.pos;
        beginposition := L^.pos;
        if L^.ustem then
            while (L^.next <> nil) and L^.ustem do
            begin
                L := L^.next;
                endposition := L^.pos;
            end;
        L := L^.next;
        if (beginposition < startposition + pagewidth)
           and (endposition >= startposition) then
        begin  (* Found an arc we should draw. *)
            while p < beginposition do
              begin  (* Space over to proper position. *)
                  for w := 1 to width do cout (' ');
                  p := p + 1;
              end;
              if beginposition = endposition then
              begin  (* Case of one node directly below. *)
                  for w := 1 to (width div 2) do cout (' ');
                  if (H <> slines) or (vertkeylength >= 0)
                  then cout('*')
                  else cout(' ');
                  for w := 1 to (width div 2) do cout (' ');
                  p := p + 1;
              end
              else
              begin  (* Normal multi-segment arc, then. *)
                  if p = beginposition then
                  begin  (* Begin with a half segment. *)
                      for w := 1 to (width div 2) do
                          cout (' ');
                      for w := (width div 2) to width-1 do
                          cout ('*');
                      p := p + 1;
                  end;
                  while (p < endposition) and
                        (p < startposition + pagewidth) do
                  begin  (* Connect to the end segment. *)
                      for w := 1 to width do cout ('*');
                      p := p + 1;
                  end;
                  if p < startposition + pagewidth then
                  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 *)
```

Three Proposals for Extending Pascal

By R.D. Tennent
Computing and Information Science
Queen's University
Kingston, Canada, K7L3N6

These three proposals have been submitted to the American Pascal Extensions Task Group and to the Canadian Pascal Working Group for consideration as extensions to ISO Pascal. The three proposals deal with separate issues and could be combined without difficulty.

The terminology and syntactic notation are those of the ISO Pascal standard. References are given only for the examples. References for and further explanation of the concepts may be found in the autor's *Principles of Programming Languages*, Prentice-Hall International, London (1981).

The Where-Clause: A Proposed Extension to Pascal

By R.D. Tennent
Computing and Information Science
Queen's University
Kingston, Canada, K7L3N6
613-547-2645

1. Introduction

A common and justified criticism of Pascal is that large programs are difficult to read. This is in part because the block bodies in a program must occur in a "bottom-up" order. The main program body (i.e., the highest-level code) appears at the very end of the program. In general, high-level code always follows the code for the lower-level procedures that it uses. The first code encountered in reading a program is that for a procedure at the lowest level of abstraction.

A further difficulty is that type, constant and variable declarations in a block can be separated from their uses by the code for procedure definitions in that block and code for all lower-level contained blocks. In large programs, the definition of an identifier can be tens of pages away from the code in which it is used.

2. The Where-Clause

The defects described above can be corrected in a very straightforward way: allow a procedure-and-function-declaration-part to follow the statement-sequence of the statement-part of a block. This is termed a *where-clause*.

```
block = label-declaration-part
        constant-definition-part
          type-definition-part
            variable-declaration-part
              procedure-and-function-declaration-part
                statement-part

statement-part = "begin"
                   statement-sequence
                   [where-clause]
                 "end"

where-clause  =  "where" procedure-and-function-declaration-part
```

The procedure-and-function-declaration-part of a where-clause is to be interpreted exactly as if it were placed immediately following the usual procedure-and-function-declaration-part of the enclosing block, except that the rule of forward-declaration before use is enforced. More formally, if D_1 and D_2 are definitions and S is a statement-sequence, then a block of the form

```
D1
begin
  S
where
  D2
end
```

shall be equivalent to

```
D1
procedure I; begin S end;
D2
begin I end
```

where I is any identifier not used in the original block.

Example 1 Partition sort, after Wirth's Algorithms + Data Structures = Programs, program 2.10.

```
procedure QuickSort;
  procedure Sort(left,right:index); forward;
    {sorts A[left..right]}
  begin {QuickSort}
```

```
    Sort(1,n)
  where
    procedure Sort{(left,right : index)};
      var i,j : index;
      procedure Partition; forward;
        {Partition computes i>j and permutes A[left..right] so
         that A[j+1..i-1] = x, A[left..i-1] ≤ x ≤ A[j+1..right]}
      begin {Sort}
        Partition;
        if left<j then Sort(left,j);
        if i<right then Sort(i,right)
      where
        procedure Partition;
          var x : item;
          procedure Exchange(var p,q : item); forward;
            {Exchange swaps the values in p and q}
          begin {Partition}
            i := left; j := right;
            x := A[(left + right) div 2];
            repeat
              while A[i]<x do i := i+1;
              while A[j]>x do j := j-1;
              if i≤j then
              begin
                Exchange(A[i],A[j]);
                i := i+1;
                j := j-1
              end
            until i>j
          where
            procedure Exchange{(var p,q : item)};
              var w : item;
              begin w := p; p := q; q := w end {Exchange};
          end {Partition};
      end {Sort};
  end {QuickSort};
```

Example 2 PL/0 compiler, after Wirth's Algorithms + Data Structures = Programs, program 5.6.

```
program PL0(input, output);
label 99;
const ...{global constants}...;
type ...{global types}...;
var ...{global variables}...;
procedure Error(n : integer); forward;
procedure Getsym; forward;
procedure Block(lev,tx : integer; fsys : symset); forward
procedure Interpret; forward;
begin {Main Program}
  ...{initialization of global variables}...
  Getsym;
  Block(0,0,[period]+declbegsys+statbegsys);
  if sym ≠ period then Error(9);
  if err=0 then Interpret else write('ERRORS IN PL/0 PROGRAM');
99: writeln
  where
    procedure Error{(n : integer)};
      begin writeln(' ****', ' ':cc-1, '^', n:2) end {Error};
    procedure Getsym;
      var i,j,k : integer;
      procedure Getch; forward;
      begin {Getsym}
      while ch = ' ' do Getch;
      ...{rest of Getsym}...
    where
      procedure Getch;
        begin {Getch} ... end {Getch};
    end {Getsym};
  procedure Block{(lev,tx : integer; fsys : symset)};
    ...{declarations for Block}...;
    begin {Block}
      ...{code for Block}...
    where
      ...{definitions of procedures used in Block}...
    end {Block};
  procedure Interpret;
    ...{declarations for Interpret}...;
    begin {Interpret}
      ...{code for Interpret}...
    where
      ...{definitions of procedures used in Interpret}...
    end {Interpret};
end {Main Program}.
```

3. Implementation

The where-clause may be implemented in essentially the same way that forward declarations in standard Pascal are implemented. The possibility of placing procedure definitions after their uses exists in many other languages, including FORTRAN, PL/I and Ada. The idea of the where-clause is due to P.J. Landin.

4. Summary

The proposed where-clause would provide two significant improvements in program readability. Firstly, it would allow a programmer to organize nested procedures in a top-down way that is often more natural to read than the bottom-up organization currently possible in Pascal. Secondly, it would allow declarations of identifiers and their uses to be much closer together than is currently possible. These advantages become more and more apparent and significant as program size and structural complexity increase.

The only costs associated with this proposal are that it would be necessary to add a new reserved word (''where'') and to increase very slightly compiler size in order to parse the new clause.

Proposals for Improved Exception-Handling in Pascal

By R.D. Tennent
Computing and Information Science
Queen's University
Kingston, Canada, K7L3N6
613-547-2645

1. Introduction

A test that indicates that an alternative computational approach must be followed and that cannot be efficiently or conviently tested before starting the computation containing the test is known as an *exceptional condition*. The alternative computation is known as the *exception-handler*. An example of an exceptional condition is overflow of a primitive numerical operation. Usually it is impractical to test for overflow before attempting the operation, but if an overflow occurs, an alternative computational path must be followed. An exception is not necessarily an "error".

The facilities for exception-handling in standard Pascal are not quite adequate. We shall describe what is currently possible, and propose relatively modest extensions to improve exception-handling for (i) programmer-defined procedures, (ii) language-defined ("required") procedures, and (iii) primitive operations.

2. Programmer-Defined Procedures

The scheme in Example 1 illustrates the most appropriate method available in standard Pascal for exception-handling. The advantages of the scheme are as follows.

(i) Code for exception-handling is separate from the code for the unexceptional cases.

(ii) It is possible to pass information to an exception handler by using the parameter-list of the procedure.

(iii) Although gotos are used, both the jumps and their destinations are at the ends of blocks only.

(iv) It is possible to invoke exception handlers defined in any enclosing block.

Example 1 Using procedures and goto statements for exception-handling.

```
label 999;
procedure ExceptionHandler(...);
  begin
    ...{computation for exceptional cases}...;
    goto 999
  end;
 :
begin {computation for unexceptional cases}
  :
   ...if...{test for exceptional condition}..
       then ExceptionHandler(...);
  :
999: end
```

If an exceptional condition must be tested in a called procedure that is defined outside the block in which the exception-handler is defined, then the exception-handler may be passed to it as a procedural parameter, as in Example 2.

Example 2 Using an exception-handling parameter.

```
procedure External(...; procedure Error(...));
  begin
    :
    ...if...{test for error}...
        then Error(...);
    :
  end;
 :
  label 999;
  procedure ExceptionHandler(...);
    begin
      ...{computation for exceptional cases}...;
      goto 999
    end;
   :
  begin {computation for unexceptional cases}
    :
      External(..., ExceptionHandler);
    :
999: end
```

We conclude that well-structured exception-handling in programmer-defined procedures is possible in standard Pascal. However, the current language does not provide specialized constructs to encourage this approach. It would be desirable to avoid use of label declarations, goto statements and numeric labels. This is the same design philosophy that led to the inclusion of the if-then-else, case, while, repeat and for control structures in Pascal.

We propose to extend Pascal by providing *exits*, which are similar to conventional procedures, but always "return" to the end of the block in which they are defined (rather than to their calls). Example 3 shows how the exception-handling schemes of Examples 1 and 2 can be expressed using exits. Exits may be implemented in the same way that the combination of non-local jumps and procedures is in Pascal, except that a dynamic "return-link" is not needed. The concept of block-exiting procedures is due to P.J. Landin.

```
procedure-and-function-declaration-part
  = { ( procedure-declaration
      | function-declaration
      | exit-declaration )    ";" }
```

```
exit-declaration = exit-heading ";" [block]

exit-heading = "exit" identifier [formal-parameter-list]

formal-parameter-section >
    value-parameter-specification
  | variable-parameter-specification
  | procedural-parameter-specification
  | functional-parameter-specification
  | exit-parameter-specification

exit-parameter-specification = exit-heading

simple-statement = empty-statement
                 | assignment-statement
                 | procedure-statement
                 | exit-statement
                 | goto-statement

exit-statement = exit-identifier [actual-parameter-list]

actual-parameter = expression | variable-access
                 | procedure-identifier
                 | function-identifier
                 | exit-identifier
```

Example 3 Using exits for exception-handling.

```
procedure External(...; exit Error(...));
 begin
   :
  ...if...{test for error}...
      then Error(...);
   :
 end;
:
 exit ExceptionHandler(...);
   begin
     ...{computation for exceptional cases}...;
  end;
  :
 begin {computation for unexceptional cases}
   :
   ...if...{test for exceptional condition}..
       then ExceptionHandler(...);
  :
   ...External(..., ExceptionHandler);
  :
 end
```

Example 4 Sift, after Wirth's Algorithms + Data Structures = Programs, program 2.7.

```
procedure Sift(l,r:index);
  var i,j:index; x:item;
  exit Sifted;
    begin a[i] := x end {Sifted};
  begin {Sift}
    i := l; j := 2*i; x := a[i];
   while j<r do
   begin
     if j<r then
        if a[j]>a[j+1] then j := j+1;
     if x<a[j] then Sifted;
        a[i] := a[j]; i := j; j := 2*i
     end;
     a[i] := x
  end {Sift};
```

Example 5 PL/0 compiler, after Wirth's Algorithms + Data Structures = Programs, program 5.6.

```
program PL0(input, output);
  :
exit Incomplete;
  begin writeln('PROGRAM INCOMPLETE.') end {Incomplete};
exit TooLong;
  begin writeln('PROGRAM TOO LONG.') end {TooLong};
  :
        ...if eof(input) then Incomplete;
  :
        ...if cx>cxmax then TooLong;
  :
begin {main program}
  :
end.
```

Example 6 Hash table look-up, after Knuth's "Structured programming with go to statements", Computing Surveys, 6, pp. 261-301, Example 3a.

```
var i:1..m;
exit Insert;
  begin
    A[i] := x; B[i] := 1; count := count+1;
    if count=m then HashTableFull
  end {Insert};
begin
  i := hash(x);
  while A[i] ≠ x do
  begin
    if A[i]=0 then Insert;
    if i>1 then i := i-1 else i := m;
  end;
  B[i] := B[i]+1
end
```

3. Required Procedures

Exception-handling is not an issue for most of the "required" procedures and functions in Pascal. Any possible failure condition can easily be tested before calling them. The only required procedure for which this is not true is Read(f,x) when f is a text file and x is an arithmetic variable. If there is a syntactic error in reading a signed-integer or signed-number from the file, the Pascal standard specifies that the execution is in error, and most implementations handle this by aborting program execution. This is a serious design error because it is essential that user-oriented software be able to recover gracefully from trivial input errors.

We propose adding the following required procedures to Pascal:

```
procedure ReadInteger(var f:text; var i:integer;
                      exit IllegalSyntax);

procedure ReadReal(var f:text; var r:real;
                   exit IllegalSyntax);
```

(An alternative approach would be to add a single required procedure ReadNumber(f,x,IllegalSyntax) whose second parameter may be either integer or real.) It is not impossible to define procedures similar to these in Pascal, but, for real numbers at least, the code is too formidable and machine-dependent for this to be a practical solution. See program 1.3 in Wirth's *Algorithms + Data Structures = Programs*. It is also possible to use Read to do the conversion after first checking the syntax of the input, using a temporary file as a buffer for the characters that are checked. But on most implementations the overhead of file creation would make this approach too inefficient.

Example 7 Handling exceptions when reading numbers.

```
procedure ReadSpeed(var s:speed; exit Failure);
var count:0..4; {Retry up to four times; then call Failure.}
exit TryAgain;
  begin
    if count=4 then Failure; count := count+1;
    writeln(output,'Incorrect syntax for signed-number. Try again.');
    ReadReal(input,s,TryAgain)
  end {TryAgain};
begin {ReadSpeed}
  count := 0;
  writeln(output,'Enter the speed as a signed-number.');
  ReadReal(input,s,TryAgain)
end {ReadSpeed};
```

4. Primitive Operations

Floating-point overflow and underflow are the only failure conditions for primitive operations that cannot reasonably be tested before applying the operations. As in the preceding section, it would be possible to add required functions with exception-handling parameters, such as

product(x,y,ovflow,unflow)

But such functions would be so much less convenient to use than the usual infix operators that this solution is impractical.

We propose adopting the following rule: an overflow or underflow shall result in calling an exit called 'Ovflow' or a procedure called 'Unflow', respectively. If, in the case of an underflowing operation, Unflow allows control to return, then the result of the operation is taken to be zero. If the overflowing or underflowing operation does not occur in the scope of a programmer-defined Ovflow or Unflow exception-handler, then the effect shall be equivalent to calling

```
exit Ovflow;
    begin {Aborts program execution} end;
```

or

```
procedure Unflow;
    begin {null} end;
```

respectively. The symbols 'Ovflow' and 'Unflow' shall be reserved in the sense that they are allowed only as the identifier of a (parameterless) exit-heading or procedure-heading, respectively, or as a corresponding actual-parameter. Note that defining Ovflow or Unflow affects only operations in the scope of the definition, and not necessarily in procedures called in that scope.

Example 8 Handling overflow and underflow.

```
procedure P(...;exit Ovflow);
  procedure Unflow;
    begin writeln('Underflow in procedure P') end;
  begin
    ...{Overflow here invokes parameter Ovflow.
       Underflow here prints a message but computation
       continues using a result of zero.}...
  end {P};
```

5. Summary

Three proposals have been made to improve exception-handling in Pascal.

(i) The exit concept has been proposed to permit well-structured exception-handling without the use of labels and gotos.

(ii) Additional required procedures for reading numbers from text files have been proposed to permit recovery from syntax errors in input files.

(iii) Conventions for handling numerical overflow and underflow have been proposed to permit recovery from machine traps while still allowing use of infix notation for arithmetic operations.

These relatively modest extensions would seem to be both necessary and sufficient to provide adequate exception-handling in Pascal. The very complex and poorly-designed exception-handling facilities in languages such as PL/I and Ada are neither necessary nor desirable, and should not be imitated in Pascal.

The Definition Block: A Proposed Extension to Pascal

By R.D. Tennent
Computing and Information Science
Queen's University
Kingston, Canada, K7L3N6
613-547-2645

1. Introduction

It is widely recognized that the block structuring facilities currently available in Pascal are inadequate to allow secure abstraction from data representations. Some rather complex proposals for extending Pascal have been made and several newer languages have addressed this issue.

The present proposal is notable for its compatibility with the concepts of Pascal. Rather than introducing new concepts having possibly complex interactions with the existing language, the facilities proposed are just straightforward generalizations of the block, record and procedure structures already present in the language.

It will be convenient to assume the existence of a variable initialization facility; however, its design does not significantly affect this proposal. We start with the following syntax; the only new feature is an initialization-part.

```
block = definitions-part
        statement-part

definitions-part = definitions

definitions = label-declaration-part
              constant-definition-part
                type-definition-part
                  variable-declaration-part
                    initialization-part
                      procedure-and-function-declaration-part

initialization-part = ["initial" block ";"]
```

Example 1 A block with an initialization part.

```
type natnum = 0..maxint;
var
  count: array[index] of natnum;
  totalcount: natnum;
initial
  var i: index;
  begin
    for i := 0 to n do count[i] := 0;
    totalcount := 0
  end {initial};
procedure Tabulate(x: index);
  begin
    if x>n then error(x);
    count[x] := count[x] + 1
    totalcount := totalcount + 1
  end {Totalcount};
function Frequency(x: index): real;
  begin
    if x>n then error(x);
    Frequency := count[x]/totalcount
  end {Frequency};
begin
  ...Tabulate(w)...Frequency(w)...
end;
```

The definitions-part of the block in Example 1 defines the representation of a histogram. This is correctly accessed in the statement-part of the block by using procedure Tabulate and function Frequency. However, identifiers 'count' and 'totalcount' are *also* accessible in the body of the block, and this is undesirable. But Pascal does not allow identifier visibility to be controlled as required (except by using procedural parameters in a rather complex and unreadable way).

2. The Definition Block

The concept used in Pascal to control visibility of identifiers is the block. Identifiers defined in the definitions-part of a block can only be used within that block. The present proposal is based on the observation that it is not essential to the block concept that the bodies of blocks be *statements*. For example, in several programming languages block bodies are *expressions*. We propose to add to Pascal a form of block whose body consists of *definitions* and may be used wherever a conventional set of definitions is usable:

```
definitions = label-declaration-part
              constant-definition-part
                type-definition-part
                  variable-declaration-part
                    initialization-part
                      procedure-and-function-declaration-part
            | definition-block

definition-block = "private" private-definitions-part
                   "within" public-definitions-part

private-definitions-part = definitions

public-definitions-part = definitions
```

Example 2 A definition-block.

```
private
  type natnum = 0..maxint;
  var
```

```
    count: array[index] of natnum;
    totalcount: natnum;
  initial
    var i: index;
    begin
     for i := 0 to n do count[i] := 0;
      totalcount := 0
    end {initial};
within
  procedure Tabulate(x: index);
    begin
      if x>n then error(x);
      count[x] := count[x] + 1
      totalcount := totalcount + 1
    end {Totalcount};
  function Frequency(x: index): real;
    begin
      if x>n then error(x);
      Frequency := count[x]/totalcount
    end {Frequency};
```

If the definition-block in Example 2 were used as the definitions-part of a conventional block, then identifiers 'Tabulate' and 'Frequency' would be accessible as usual in the statement-part, whereas identifiers 'natnum', 'count' and 'totalcount' could be used only within the definition-block itself. Definition-blocks thus provide control over identifier visibility for definitions in essentially the same way that conventional blocks do for statements. It is simpler, more readable and more convenient than other approaches to scope control, such as lists of "exported" identifiers or "tags" on individual definitions. The concept of the definition-block is due to P.J. Landin.

The implementation of definition blocks is very simple. A compiler must ensure that the definitions of the private part are not visible outside of the definition-block, but are visible in the public part. The externally-visible effects of the definition-block are those of the public part alone. However, at run-time, the activation record created by elaborating the private part must be retained with the activation record for the public part. This is because procedures defined in the public part may refer to variables declared in the private part.

3. Generalized Records

Although the definition-block is the basis of this proposal, it is of rather limited utility by itself. Two simple generalizations of Pascal concepts will allow creation and naming of multiple instances of activation records from definitions, and parameterization of definitions.

The most convenient way of providing multiple instances in Pascal is to take advantage of the similarity between activation records (created by elaborating definitions) and values of record-types. We propose generalizing record-types as follows:

record-type = "record" (field-list | definitions) "end"

The use of definitions in place of a conventional field-list allows records to have components that are constants, types and procedures, as well as variables. Most importantly, if the definitions part of a record-type is a definition-block, there can be variable components that are private but are indirectly accessible via public procedure components.

Example 3 A block that uses a generalized record type.

```
var MaleWeights, FemaleWeights:
      record
        private
          type natnum = 0..maxint;
          var
            count: array[index] of natnum;
            totalcount: natnum;
          initial
            ...
        within
          procedure Tabulate(x: index);
            ...
          function Frequency(x: index): real;
            ...
      end;
begin
  ...MaleWeights.Tabulate(mw)...MaleWeights.Frequency(mw)...
  ...with Femaleweights do
     begin...Tabulate(fw)...Frequency(fw)...end...
end;
```

In Example 3, two histograms are created by declaring variables MaleWeights and FemaleWeights, using a record-type containing the definition-block of Example 2 instead of a field-list. Each variable is allocated a separate storage area for its variable components within the activation record for the block. It is also possible to define a name for the new type in the usual way.

Example 4 Naming a generalized record-type.

```
type WeightHistogram = record
                         private
                           ...
                         within
                           ...
                       end;
var MaleWeights, FemaleWeights: WeightHistogram;
procedure PrintWeightFrequencies(var wh: WeightHistogram);
    ...wh.Frequency(w)...
begin
  ...PrintWeightFrequencies(MaleWeights)...
  ...PrintWeightFrequencies(FemaleWeights)...
end
```

In Example 4, the type-identifier has been used to specify the type of a formal var parameter. Value parameters (and assignment) of generalized record-types would be possible but are not recommended (cf. file-types in standard Pascal).

4. Classes

It is desirable to be able to parameterize definitions or to create more than one new-type from the same set of definitions. For these purposes we propose adding a form of "procedure" whose definition body and calls are generalized record-types. These are termed *classes*, after the similar concept in SIMULA.

```
definitions =
      label-declaration-part
        constant-definition-part
          class-definition-part
```

```
            type-definition-part
              variable-declaration-part
                initialization-part
                  procedure-and-function-declaration-part
      | definition-block

class-definition-part = { class-definition ";" }

class-definition = class-heading ";" "record" definitions "end"

class-heading = "class" identifier [formal-parameter-list]

record-type = "record" ( field-list | definitions ) "end"
            | class-identifier [actual-parameter-list]
```

Example 5 Definition and use of a class.

```
class Histogram(n:index; procedure error(x:index));
  record
    private
      type natnum = 0..maxint;
      var
        ...
      initial
        ...
    within
      procedure Tabulate(x:index);
        ...
      function Frequency(x:index):real;
        ...
  end {Histogram};
var
  MaleWeight, FemaleWeight: Histogram(nw,WeightError);
  MaleHeight, FemaleHeight: Histogram(nh,HeightError);
begin ... end;
```

When a class is called, the actual parameters are evaluated, the formal-parameters are bound, and a new type is created. The definitions in the body are elaborated when the type is used in a variable declaration.

5. Summary

Flexible, convenient and easily-implemented facilities to control identifier visibility in definitions and abstract from data representations can be added to Pascal simply by generalizing the blocks, records and procedures already present in the language.

Dear Mr. Cichelli,

I am enclosing a check for $5.00 to cover costs for the mailing of issues 20 and 21, which I have not received.

I suspect it is the fault of our local post office, which has failed to deliver several other pieces of mail recently, and has returned them to the sender.

If $5.00 is insufficient for two issues, please let me know, as I value PN quite highly. I consider it a model, along with Dr. Dobbs, against which to measure other computer-related publications. In particular, I am appreciative of a journal which does not condescend or patronize its readers. There is a sense of dignity and scientific humility in the editorial approach which is apparently contagious, affecting those who write to you, and those who publish articles.

I would very much like to have the software tools available on magnetic tape, and would support a price in the 100 dollar range. Currently, I have access to an IBM 370, and can accept either EBCDIC or ASCII input tapes in a wide variety of BPI values, labeled or unlabeled. I suspect that any format you set up will include an acceptable version for this application.

I am currently employed by CTB/McGraw-Hill in Monterey, Calif., where I have recently instituted a software tools special interest group, whose purpose is to examine the daily operations of the company, whether directly in DP or not, and to design, develop and test a set of software tools for practical use by any division of the company. Though these tools will likely be originally written in PL/I, Fortran or Cobol, (our principal languages), there is sufficient interest in the projects that we may see a number of them re-written in Pascal. I will try to keep you informed of our progress, and to submit any tested tools for your consideration.

Thanks again for a wonderful publication.

Charles Frankle
607 2nd St.
Pacific Grove CA 93950

Dear Rick,

I want to complain about an item in *Pascal News #22-23* on page 38, which I think should not have been published. This is a function in the Applications section named OPTIONS which is an altered version of a function I wrote named OPTION used to return control-statement option settings.

My objections are as follows:

1. The code is copyright University of Minnesota, but no one asked our permission to publish it.
2. The code is for the CDC-6000 Pascal implementation only — it is of insignificant interest to the Pascal community at large.
3. The code is published without any explanation of how it relates in time and space to anything else in the universe!
4. The code is a perfect example of bad Pascal programming (my own) which I not only find embarrassing, but also goes against the philosophy of the Applications section, which is to publish only good examples.
5. The code is in the wrong character set. It is in un-reconverted CDC 63 SCIENTIFIC, which prints → for pointers and ↑ for quotes (the correct characters would be ↑ and ≡ respectively.

Spike Leonard of Sandia National Laboratories (an AT&T subsidiary) at Livermore, California, submitted the extended version to accept 1-10-character option names instead of 1-character option names and to return equivalenced parameter values (in CDC operating-system terminology). Unfortunately my name is more prominent than his in the comments and should be the other way around. I regret that Spike didn't send us here in Minnesota this extended version for inclusion into the CDC-6000 Pascal compiler system that we maintain. The first time I ever saw the code was in *Pascal News #22-23*! Also, I realize I am listed as co-editor of the Applications section, but Richard Cichelli, the other co-editor has actually done nearly all of the work on the section since *Pascal News #16*.

I called Spike and had a good talk with him, and of course he was trying to help the Pascal cause. I enclose the original code so that readers can discern Spike's contribution. Perhaps this all goes to show me that I shouldn't have written bad code to begin with!

Rick, I want to congratulate you on the fine job you have done in keeping *Pascal News* alive during the last 2 difficult years in the face of many obstacles: your full-time job, the ever-increasing number of subscribers, the dwindling amount of volunteer help, the lack of support of your organization, and the difficulties caused by the collapse of PUG(Europe) in the UK, which is only now being resolved. I could *not* have done as well as you have done if I had continued as editor — as you know, I was on the verge of collapse.

In issues # 17-23 which you have edited, the substantive information about applications, compiler validation suite reports, correspondence from subscribers, and drafts of the ISO Pascal Standard continues. I think that's important, and I realize that you haven't been able to keep up detailed information on news tidbits as you would have liked to do.

I wish you good luck in finding a successor. Like I wrote in my farewell, *Pascal News* may not last past 1982.

Sincerely,
Andy

```
(**       OPTION - RETURN CONTROL STATEMENT OPTION SETTING.
*         COPYRIGHT (C) UNIVERSITY OF MINNESOTA  - 1978.
*         A. B. MICKEL.       77/06/02.
*
*         SEE THE PASCLIB WRITEUP FOR EXTERNAL DOCUMENTATION.
*)

FUNCTION OPTION(NAME: CHAR; VAR S: SETTING): BOOLEAN;

CONST
   CSADDRESS = 70B (*CONTROL STATEMENT ADDRESS*);

TYPE
   CSIMAGEP = RECORD CASE BOOLEAN OF
                FALSE: (A: INTEGER);
                TRUE: (P: ^LOWCORE)
              END;
   LOWCORE = PACKED ARRAY[1..80] OF CHAR;

VAR
   CSIMAGE: CSIMAGEP,
   I: INTEGER (*INDEX IN CSIMAGE*);
   FOUND: BOOLEAN;

BEGIN (*OPTION*)
   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;
         IF CSIMAGE.P^[I] = NAME THEN
            BEGIN FOUND := TRUE;
            I := I + 1;
            S.SWITCH := CSIMAGE.P^[I] IN ['+', '-', '='];
            IF S.SWITCH THEN S.ONOFF := CSIMAGE.P^[I]
            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
         ELSE WHILE NOT (CSIMAGE.P^[I] IN [',', ')', '.']) DO
                 I := I + 1,
      UNTIL (CSIMAGE.P^[I] IN [')', '.']) OR FOUND;
   OPTION := FOUND
END (*OPTION*);
```

IMPLEMENTATION NOTES ONE PURPOSE COUPON

. **DATE**

1. **IMPLEMENTOR/MAINTAINER/DISTRIBUTOR** *(* Give a person, address and phone number. *)*

2. **MACHINE/SYSTEM CONFIGURATION** *(* Any known limits on the configuration or support software required, e.g. operating system. *)*

3. **DISTRIBUTION** *(* Who to ask, how it comes, in what options, and at what price. *)*

. **DOCUMENTATION** *(* What is available and where. *)*

5. **MAINTENANCE** *(* Is it unmaintained, fully maintained, etc? *)*

6. **STANDARD** *(* How does it measure up to standard Pascal? Is it a subset? Extended? How. *)*

7. **MEASUREMENTS** *(* Of its speed or space. *)*

8. **RELIABILITY** *(* Any information about field use or sites installed. *)*

9. **DEVELOPMENT METHOD** *(* How was it developed and what was it written in? *)*

10. **LIBRARY SUPPORT** *(* Any other support for compiler in the form of linkages to other languages, source libraries, etc. *)*

(FOLD HERE)

PLACE
POSTAGE
HERE

Bob Dietrich
M.S. 92-134
Tektronix, Inc.
P.O. Box 500
Beaverton, Oregon 97077
U.S.A.

(FOLD HERE)

NOTE: Pascal News publishes all the checklists it gets. Implementors should send us their checklists for their products so the thousands of committed Pascalers can judge them for their merit. Otherwise we must rely on rumors.

Please feel free to use additional sheets of paper.

New Subscription
Re New
Back Issue

_____ *COUPON* _____

(Jan. 83)

Pascal News
2903 Huntington Road
Cleveland, Ohio 44120

** 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	$20	£12	DM40	A$16
☐ Renew my subscription for:	3 years	$40	£24	DM80	A$32

☐ Send Back Issue Set(s)

☐ 1	☐ 2	☐ 3	☐ 4
9-12	13-16	17-20	21-23

- Issues 1 . . 8 (January, 1974 — May 1977) are *out of print.*
- Issues 9 . . 12, 13 . . 16, & 17 . . 20, 21 are available from PUG(USA) all for $15.00 a set, and from PUG(AUS) all for $A15.00 a set.

☐ 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 USER GROUP?

- Membership is open to anyone: Particularly the Pascal user, teacher, maintainer, implementor, distributor, or just plain fan.
- Please enclose the proper prepayment (check payable to "Pascal User's Group"); 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 *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 :* join through PUG(UK) : Pascal Users Group, % Shetlandtel, Walls, Shetland, ZE2 9PF, United Kingdom.
- *Australasian Region* (Australia, East Asia — incl. India & Japan): PUG(AUS). Pascal Users Group, % Arthur Sales, 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?

- Back issues will have a price rise to $25 on July 83
- 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, 21 . . 23 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: $10 each — PUG(USA); and $A10.00 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 15.5 cm. wide) form.
- All letters will be printed unless they contain a request to the contrary.

APPLICATION FOR LICENSE TO USE VALIDATION SUITE FOR PASCAL

Name and address of 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

Facts about Pascal, THE PROGRAMMING LANGUAGE:

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

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

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

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

The standard reference and tutorial manual for Pascal is:

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

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

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

Remember, Pascal User's Group is each individual member's group. We currently have more than 3500 active members in more than 41 countries. This year *Pascal News* is averaging more than 100 pages per issue.

FINIS