program Referencer(input,output);  {$D2,S+ }

PASCAL PROCEDURAL CROSS-REFERENCER


DEVELOPMENT

This program is a software tool developed from a prototype by A.J.Currie at the University of Southampton, England. The prototype of 231 lines of source text was used firstly as a basis for extensions, and then rewritten to assure correctness by A.H.J.Sale, on leave from the University of Tasmania and then also at the University of Southampton. The current version was stabilized at 1979 December 4; the development time being estimated at 4 man-days from prototype to production.

PURPOSE

The program reads Pascal source programs and produces two tables as output. These tables are procedural documentation and cross-references. One documents all procedure or function headings in a format that illustrates lexical nesting. The other tables gives the locations of heading, block, and body for each procedure and function, and what procedures and functions it immediately calls.

There is a User Manual for this program; if it has not been provided with your installation write to:

Department of Information Science
University of Tasmania
P.O.Box 252C, G.P.O. Hobart
Tasmania 7001

and ask for the Technical Report on Referencer, if it is still available. The program is written to be portable and is believed to be in Standard Pascal.

Permission is granted to copy this program, store it in a computer system, and distribute it, provided that this header comment is retained in all copies.

const

{ This constant is the number of significant characters kept in the identifier entries. It can readily be changed. It is not advised that it be reduced below 10 (reserved words get to 9). }
SigCharLimit = 16;

{ This must always be (SigCharLimit - 1). It is used simply to reduce the set range to have a lower bound of 0, not 1. }
SetLimit = 15;
{
  This constant is used to convert upper-case letters to lower-case
  and vice-versa. It should be equal to ord('a') - ord('A').
}
UCLCdisplacement = 32;
{
  This constant determines the size of the input line buffer.
  The maximum acceptable input line is one smaller because a sentinel
  space is appended to every line.
}
LineLimit = 200;
{
  This constant determines the maximum width of the printing of the
  second cross-reference table. The program deduces how many names
  will fit on a line.
}
LineWidth = 132;
{
  This determines the indentation of the lex-levels.
}
Indentation = 4;
{
  These constants are used for the sketchy syntax analysis.
  They are collected here so that their lengths may be altered if
  SigCharLimit is altered.
}
Sprogram = 'program         ';
Sprocedure = 'procedure       ';
Sfunction = 'function        ';
Slable = 'label           ';
Sconst = 'const           ';
Stype = 'type            ';
Svar = 'var             ';
Sbegin = 'begin           ';
Scase = 'case           ';
Send = 'end             ';
Sforward = 'forward       ';
Spaces = '                ';


type
Natural = 0..maxint;
Positive = 1..maxint;
SixChars = packed array[1..6] of char;
SigCharRange = 1..SigCharLimit;
SetRange = 0..SetLimit;
PseudoString = packed array [SigCharRange] of char;
StringCases = set of SetRange;

LineSize = 1..LineLimit;
LineIndex = 0..LineLimit;
SetOfChar = set of char;
ProcKind = (FwdHalf,AllFwd,Shortform,Formal,Outside,NotProc);
PtrToEntry = ^ Entry;
ListOfUsages = ^ UsageCell;
PtrToStackCell = ^ StackCell;
TokenType = (OtherSy,NameSy,LParenSy,RParenSy,ColonSy,
  SemiColSy,PeriodSy,AssignSy,SubRangeSy);

{ This type represents a procedure or function identifier found
during processing of a program. The fields are used as follows:
- procname & caseset = representation of name
- linenumber = where heading starts
- startofbody = where begin of statement-part starts
- forwardblock = where forward-declared block starts
- status = kind or status of name
- left, right = subtrees of the scope-level tree
- before, after = subtrees of the supertree
- calls = a list of the procedures this calls
- localtree = the scope tree for the interior

Entry =
   record
      procname : PseudoString;
      caseset : StringCases;
      linenumber : Natural;
      startofbody : Natural;
      before, after : PtrToEntry;
      localtree : PtrToEntry;
      calls : ListOfUsages;
      case status : ProcKind =
         FwdHalf, Shortform, Formal, Outside, NotProc:
         ();
         AllFwd:
         ( forwardblock: Natural )
      end;
   end;

{ This type records an instance of an activation of a procedure or function. The next pointers maintain an alphabetically ordered list; the what pointer points to the name of the activated code. }
UsageCell =
   record
      what : PtrToEntry;
      next : ListOfUsages
   end;

{ This type is used to construct a stack which holds the current lexical level information. }
StackCell =
   record
      current : PtrToEntry;
      scopetree : PtrToEntry;
      substack : PtrToStackCell
   end;

var
  lineno : Natural;
  chno : LineIndex;
  total : LineIndex;
  depth : Natural;
  level : -1..maxint;
  pretty : Natural;

  adjustment : (First, Other);
  movement : integer;

  printflag : boolean;
  errorflag : boolean;

  ch : char;

  token : tokentype;
symbol       : PseudoString;
symbolcase   : StringCases;
savesymbol   : PseudoString;
line         : array[LineSize] of char;
superroot    : PtrToEntry;
stack        : PtrToStackCell;

{ The remaining variables are pseudo-constants. }
alphabet     : SetOfChar;
alphanums     : SetOfChar;
uppercase     : SetOfChar;
digits       : SetOfChar;
usefulchars  : SetOfChar;
namesperline : Positive;

procedure PrintLine;
var
    i : LineSize;
begin
    write(output, lineno:5, '    ');
    i := 1;
    { Is this the first time in a run or not? }
    if adjustment = First then begin
        { Ignore any leading spaces there happen to be. }
        while (i < total) and (line[i] = ' ') do
            i := succ(i);
        { Compute the adjustment needed for other lines. }
        movement := (level * Indentation) - (i - 1);
        adjustment := Other;
        { Insert any necessary indentation }
        if level > 0 then
            write(output, ' ': (level*Indentation))
        end else begin
            { It wasn't the first time, so try to adjust this
            line to align with its mother. }
            if movement > 0 then begin
                write(output, ' ':movement)
            end else if movement < 0 then begin
                while (i < total) and (line[i] = ' ') and
                    (i <= - movement) do begin
                    i := succ(i)
                end
            end
        end;
    end;
    { Write out the line. }
    while i < total do begin
        write(output, line[i]);
        i := succ(i)
    end;
    writeln(output)
end; { PrintLine }

procedure Error(e: Positive);
{ This procedure is the error message repository. }
begin
    errorflag := true;
    write(output, 'FATAL ERROR - ');
    case e of
        1: write(output, 'No "program" word');
2: write(output, 'No identifier after prog/proc/func');
3: write(output, 'Token after heading expected');
4: write(output, 'Lost ".", check begin/case/ends');
5: write(output, 'Same name, but not forward-declared')
end;
{ We shall print the offending line too. }
writeln(output, ' - AT FOLLOWING LINE');
PrintLine
end; { Error }

procedure NextCh;
begin
  if chno = total then begin
    if printflag then
      PrintLine;
    total := 0;
    while not eoln(input) do begin
      total := succ(total);
      read(input, line[total])
    end;
  end else begin
    chno := succ(chno);
    ch := line[chno]
  end
end; { NextCh }

procedure Push(newscope: PtrToEntry);
var
  newlevel: PtrToStackCell;
begin
  new(newlevel);
  newlevel^.current := newscope;
  newlevel^.scopetree := nil;
  newlevel^.substack := stack;
  stack := newlevel;
  level := level + 1
end; { Push }

procedure Pop;
var
  oldcell: PtrToStackCell;
begin
  stack^.current^.localtree := stack^.scopetree;
  oldcell := stack;
  stack := oldcell^.substack;
  { *** dispose(oldcell); *** }
  level := level - 1
end; { Pop }

procedure FindNode(var match : Boolean;
var follow : PtrToEntry;
var thisnode: PtrToEntry);
begin
  match := false;
  while (thisnode <> nil) and not match do begin
    follow := thisnode;
    if savesymbol < thisnode^.procname then
      thisnode := thisnode^.left
else if savesymbol > thisnode^.procname then
  thisnode := thisnode^.right
else
  match := true
end; { FindNode }

function MakeEntry (mainprog: Boolean;
proc    : Boolean): PtrToEntry;
{ The first parameter is true if the name in symbol is the
program identifier, which has no scope. The second parameter
is true if the name in symbol is that of a procedure or function.
The result returned is the identification of the relevant record. }
var
  newentry, node: PtrToEntry;
  located: Boolean;

begin { PutToSuperTree }
  if superroot = nil then begin
    { Nothing in the supertree yet. }
    superroot := newnode
  end else begin
    { Seek the right place }
    FindLeaf;
    with place^ do begin
      if savesymbol < procname then
        before := newnode
      else
        after := newnode
    end
  end; { PutToSuperTree }

begin { MakeEntry }
  located := false;
  savesymbol := symbol;
  if mainprog then begin
    new(newentry);
  end else if stack^.scopetree = nil then begin
    { Nothing here yet. }
new(newentry);
   stack^.scopetree := newentry
end else begin
   ( Seek the identifier in the tree. )
   FindNode(located, node, stack^.scopetree);
   if not located then begin
      ( Normal case, make an entry. )
      new(newentry);
      with node^ do
      if symbol < procname then
         left := newentry
      else
         right := newentry
   end
end;
if not located then begin
   ( Here we initialize all the fields )
   with newentry^ do begin
      procname := symbol;
      caseset := symbolcase;
      linenumber := lineno;
      startofbody := 0;
      if proc then
         status := Shortform
      else
         status := NotProc;
      left := nil;
      right := nil;
      before := nil;
      after := nil;
      calls := nil;
      localtree := nil
   end;
   MakeEntry := newentry;
   if proc then begin
      PutToSuperTree(newentry);
      Push(newentry)
   end else begin
      ( Well, It'd better be forward or else. )
      MakeEntry := node;
      Push(node);
      if node^.status = FwdHalf then begin
         stack^.scopetree := node^.localtree;
         node^.status := AllFwd;
         node^.forwardblock := lineno
      end else begin
         Error(5)
      end
   end
end;
end; { MakeEntry }
procedure PrintTree(root: PtrToEntry);
var
   thiscell: ListOfUsages;
   count: Natural;
procedure ConditionalWrite(n: Natural;
                           substitute: SixChars);
begin
   { Write either the substitute string or a number. }
   if n = 0 then
      write(output, substitute)
   else
      write(output, n:6)
PASCAL COMPUTER LANGUAGE INFORMATION

end; { ConditionalWrite }

procedure NameWrite(p : PtrToEntry);
var
  s : SetRange;
begin
  for s := 0 to SetLimit do begin
    if s in p^.caseset then
      write(output,
        chr(ord(p^.procname[s+1])-UCLCdisplacement))
    else
      write(output, p^.procname[s+1])
  end
end; { NameWrite }

begin { PrintTree }
  if root <> nil then
    with root^ do begin
      PrintTree(before);
      writeln(output);
      write(output, linenumber: 5);
      ConditionalWrite(startofbody, '      ');
      case status of
        FwdHalf, NotProc:
          write(output, '   eh?');
        Formal:
          write(output, '   fml');
        Outside:
          write(output, '   ext');
        Shortform:
          write(output, '     ');
        AllFwd:
          write(output, '  ');
          NameWrite(root);
          write(output, ' :');
          thiscell := calls;
          count := 0;
          while thiscell <> nil do begin
            if ((count mod namesperline) = 0) and (count <> 0)
              then begin
              writeln(output);
              write(output, '  ':35, ' :')
            end;
            NameWrite(thiscell^.what);
            thiscell := thiscell^.next;
            count := count + 1
          end;
      writeln(output);
      PrintTree(after)
    end
  end;

procedure NextToken;
{ This procedure produces the next "token" in a small set of
  recognized tokens. Most of these serve an incidental purpose;
  the prime purpose is to recognize names (res'd words or identifiers).
  It serves also to skip dangerous characters in comments, strings,
  and numbers. }

procedure IgnoreComment;
0521:          { This procedure skips over comments according to the definition
0522:          in the Draft Pascal Standard. }
0523:          begin
0524:              NextCh;
0525:              repeat
0526:                  while (ch <> '*') and (ch <> '}') do
0527:                      NextCh;
0528:                  if ch = '*' then
0529:                      NextCh;
0530:              until (ch = ')') or (ch = '}');
0531:              NextCh
0532:          end; { IgnoreComment }
0533:
0534:          procedure IgnoreNumbers;
0535:          { This procedure skips numbers because the exponent part
0536:          just might get recognized as a name! Care must be taken
0537:          not to consume half of a ".." occurring in a construct like
0538:          "1..Name", or worse to consume it and treat the name as a
0539:          possible exponent as in "1..E02". Ugh. }
0540:          begin
0541:              while ch in digits do
0542:                  NextCh;
0543:              { The construction of NextCh, chno, & line ensure that
0544:              the following tests are always defined. It is to get
0545:              rid of tokens which begin with a period like .. & .) }
0546:              if (ch = '.') then begin
0547:                  if (line[chno+1] in digits) then begin
0548:                      NextCh;
0549:                      while ch in digits do
0550:                          NextCh;
0551:                  end;
0552:              end;
0553:              if (ch = 'E') or (ch = 'e') then begin
0554:                  NextCh;
0555:                  if (ch = '+') or (ch = '-') then
0556:                      NextCh;
0557:              end;
0558:          end; { IgnoreNumbers }
0559:          procedure ReadIdent;
0560:          { This procedure reads in an identifier }
0561:          var
0562:              j : Positive;
0563:          begin
0564:              token := NameSy;
0565:              symbol := Spaces;
0566:              symbolcase := [];
0567:              j := 1;
0568:              while (j <= SigCharLimit) and (ch in alphanums) do begin
0569:                  if ch in uppercase then begin
0570:                      symbol[j] := chr(ord(ch) + UCLCdisplacement);
0571:                      symbolcase := symbolcase + [j-1];
0572:                  end;
0573:                  j := j+1;
0574:              end;
0575:              while ch in alphanums do
0576:                  NextCh
0577:          end; { ReadIdent }
begin { NextToken }
    token := OtherSy;
repeat
    if ch in usefulchars then begin
        case ch of
            ')': begin
                NextCh;
                token := RParenSy
            end;
            '(' begin
                NextCh;
                if ch = '*' then begin
                    IgnoreComment
                end else begin
                    token := LParenSy
                end
            end;
            '{' begin
                IgnoreComment
            end;
            ''' begin
                NextCh;
                while ch <> ''' do
                    NextCh;
                NextCh
            end;
            '0','1','2','3','4','5','6','7','8','9':
                begin
                    IgnoreNumbers
                end;
            ':': begin
                NextCh;
                if ch = '=' then begin
                    token := AssignSy;
                    NextCh
                end else begin
                    token := ColonSy
                end
            end;
            '.': begin
                NextCh;
                if ch <> '.' then
                    token := PeriodSy
                else begin
                    token := SubRangeSy;
                    NextCh
                end
            end;
            ';': begin
                NextCh;
                token := SemiColSy
            end;
        end;
    end;
end;

'0','1','2','3','4','5','6','7','8','9':
begin
    IgnoreNumbers
end;

'': begin
    NextCh;
    if ch = '=' then begin
        token := AssignSy;
        NextCh
    end else begin
        token := ColonSy
    end
end;

'1','2','3','4','5','6','7','8','9':
begin
    IgnoreNumbers
end;

'.' begin
    NextCh;
    if ch <> '.' then
        token := PeriodSy
    else begin
        token := SubRangeSy;
        NextCh
    end
end;

';' begin
    NextCh;
    token := SemiColSy
end;

'A','B','C','D','E','F','G','H','I','J','K','L','M',
    'N','O','P','Q','R','S','T','U','V','W','X','Y','Z',
'A','B','C','D','E','F','G','H','I','J','K','L','M',
    'N','O','P','Q','R','S','T','U','V','W','X','Y','Z',
'A','B','C','D','E','F','G','H','I','J','K','L','M',
    'N','O','P','Q','R','S','T','U','V','W','X','Y','Z',


begin
  ReadIdent
end

end
end else begin
  { Uninteresting character }
  NextCh
end

until token <> OtherSy
end; { NextToken }

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

var
  at : PtrToEntry;

function NameIsInScope: Boolean;
{ This function is called during the declaration phase of a block, and has to find any procedure which gets renamed by the scope rules. }

var
  llevel        : PtrToStackCell;
  discovered    : Boolean;
  where         : PtrToEntry;

begin
  llevel := stack;
  discovered := false;
  savesymbol := symbol;
  while (llevel <> nil) and not discovered do begin
    FindNode(discovered, where, llevel^.scopetree);
    if not discovered then
      llevel := llevel^.substack
  end;
  if discovered then
    NameIsInScope := (where^.status <> NotProc)
  else
    NameIsInScope := false
end; { NameIsInScope }

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

var
  address: PtrToEntry;

procedure CrossReferencer;
{ CrossReferencer is called whenever we have a name which might be a call to a procedure or function. The only way we tell is by looking in the table to see. If it is, then the list of usages of the procedure we are in is scanned and possibly extended. }

var
  newcell : ListOfUsages;
  ptr     : ListOfUsages;
  home    : PtrToEntry;
  slevel  : PtrToStackCell;
  found   : Boolean;

procedure FindCell;
{ FindCell is used to scan a List Of Usages to determine whether the name already appears there. If not, it
leaves ptr pointing to the tail of the list so that an
addition can be made. }

var

nextptr : ListOfUsages;

begin

found := false;

nextptr := stack^.current^.calls;

if nextptr <> nil then

repeat

ptr := nextptr;

found := (ptr^.what^.procname = savesymbol);

nextptr := ptr^.next

until found or (nextptr = nil)

else

ptr := nil

end; { FindCell }

begin { CrossReferencer }

slevel := stack;

found := false;

while (slevel <> nil) and not found do begin

FindNode(found, home, slevel^.scopetree);

if not found then

slevel := slevel^.substack

end;

if found then begin

if home^.status <> NotProc then begin

FindCell;

if not found then begin

new(newcell);

if ptr <> nil then

ptr^.next := newcell

else

stack^.current^.calls := newcell;

newcell^.what := home;

newcell^.next := nil

end

end

end; { CrossReferencer }

procedure ScanForName;

{ This procedure is required to go forward until the
current token is a name (reserved word or identifier). }

begin

NextToken;

while token <> NameSy do

NextToken

end; { ScanForName }

begin { ProcessBlock }

while (symbol <> Sbegin) do begin

while (symbol <> Sbegin) and (symbol <> Sprocedure) and

(symbol <> Sfunction) do begin

ScanForName;

if NameIsInScope then begin

address := MakeEntry(false, false);

{ MakeEntry made its status NotProc }

end

end;

if symbol <> Sbegin then begin

ProcessUnit(false);

ScanForName

end

end;
(We have now arrived at the body)

depth := 1;
stack^.current^.startofbody := lineno;

while depth <> 0 do begin
  if token <> NameSy then begin
    NextToken
  end else begin
    if (symbol = Sbegin) or (symbol = Scase) then begin
      depth := depth + 1;
      NextToken
    end else if (symbol = Send) then begin
      depth := depth - 1;
      NextToken
    end else begin
      { This name is a candidate call. But first we
      must eliminate assignments to function values. }
      savesymbol := symbol;
      NextToken;
      if token <> AssignSy then begin
        CrossReferencer
      end else begin
        NextToken
      end
    end
  end
end;

procedure ScanParameters;
{ This procedure scans the parameter list because at the outer
level there may be a formal procedure we ought to know about. }
var
  which : PtrToEntry;

begin
  NextToken;
  while token <> RParenSy do begin
    if (token = NameSy) then begin
      if (symbol = Sprocedure) or
          (symbol = Sfunction) then begin
        { A formal procedural/functional parameter. }
        NextToken;
      end else begin
        which := MakeEntry(false, true);
        which^.status := Formal;
        Pop;
        NextToken;
        if token = LParenSy then begin
          { Skip interior lists. }
          ScanTillClose
        end
      end
    end else begin
      if token = RParenSy do begin
        NextToken;
      end
    end;
  end;
end else begin
  Error(2);
  NextToken
end
end else begin
  if NameIsInScope then
    which := MakeEntry(false, false);
  NextToken
end
end else begin
  NameInScope;
  NextToken
end;
end; ( ScanParameters )

begin ( ProcessUnit )
  printflag := true;
  adjustment := First;
  NextToken;
  if token <> NameSy then
    Error(2)
  else begin
    at := MakeEntry(programid, true);
    while not (token in [LParenSy, SemiColSy, ColonSy]) do
      NextToken;
    if token = LParenSy then
      ScanParameters;
    while token <> SemiColSy do
      NextToken;
    PrintLine;
    ( We have now printed the procedure heading. )
    printflag := false;
    writeln(output);
    ( Our next task is to see if there is an attached block. )
    NextToken;
    if token <> NameSy then
      Error(3)
    else begin
      if (symbol <> Slabel) and (symbol <> Sconst) and
        (symbol <> Sfunction) and (symbol <> Sprocedure) and
        (symbol <> Sbegin) then begin
        ( Bloody directive, mate. )
        if symbol = Sforward then
          at^.status := FwdHalf
        else
          at^.status := Outside;
      end;
      { ProcessUnit }
  end; ( ProcessUnit )
const
NoOfNames = 2;

type
NamesIndex = 1..NoOfNames;

var
kk : NamesIndex;
tt : array[NamesIndex] of PseudoString;
hohum: PtrToEntry;

begin
  tt[01] := 'new             ';     
  tt[02] := 'writeln         ';     
  caseset := [];
  for kk := 1 to NoOfNames do begin
    symbol := tt[kk];        
    hohum := MakeEntry(false,false);   
    hohum^.status := Outside;
  end;
end;

----------------------------------- *** }

procedure PrintHeading;
begin
  writeln(output, 'Procedural Cross-Referencer - Version S-02.02');
  writeln(output, '=============================================');
end; { PrintHeading }

begin { Referencer }
  superroot := nil;
  { Here we construct an outer-scope stack entry. This is needed
  to hold any pre-defined names. The distributed version does not
  include any of these, but they are easily provided. See the
  outlines in the code marked with *** if you want this feature. }
  new(stack);
  with stack^ do begin
    current := nil;
    scopetree := nil;
    substack := nil
  end;
  printflag := false;
  uppercase := ['A','B','C','D','E','F','G','H','I','J','K','L','M','
                'N','O','P','Q','R','S','T','U','V','W','X','Y','Z'];
  alphabet  := uppercase +
                ['a','b','c','d','e','f','g','h','i','j','k','l','m',
                 'n','o','p','q','r','s','t','u','v','w','x','y','z'];
  digits    := ['0','1','2','3','4','5','6','7','8','9'];
  alphanums := alphabet + digits { *** + ['] *** } ;
  usefulchars := alphabet + digits +
                 ['(',')',' ','
                  '',' ','
                  '',' ','
                  '',' '];
  namesperline := (LineWidth - (SigCharLimit + 21)) div 
                 (SigCharLimit + 1);
  (SigCharLimit + 1);

  { *** If you want to introduce some options, this is the place
  to insert the call to your OptionAnalyser. None is provided
  with the standard tool because the requirements vary widely
  across user environments. The probable options that might be
  provided are (a) whether pre-declared names should appear in
  the call lists, (b) how many columns are to be printed in them
  (namesperline), (c) whether underscore is permitted in identifiers,
  and perhaps whether output should be completely in upper-case
  letters. The first option (a) requires a call to BuildPreDefined


just below this point, after analysing options...

total := 0;
chno := 0;
lineno := 0;
level := -1;
errorflag := false;

{ *** BuildPreDefined; *** }

{ *** } page(output); { *** }

PrintHeading;
writeln(output, ' Line Program/procedure/function heading');
for pretty := 1 to 43 do
    write(output, '-');
writeln(output);
writeln(output);

{ Now we need to get the first token, which should be program. }
ch := ' '; { *** bug fix - JTE SSRFC U of Minn 1981-03-18 }
NextToken;
if token <> NameSy then
    Error(1)
else if symbol <> Sprogram then
    Error(1)
else begin
    ProcessUnit(true);
    { Having returned, there ought to be a period here. }
    if not errorflag then begin
        { We check all tokens that begin with a period because
          what occurs after the closing period is nothing to do
          with us. }
        if (token <> PeriodSy) and (token <> SubRangeSy) then
            Error(4)
        else begin
            adjustment := First;
            PrintLine
        end
    end
end;
{ Completed Phase One - now for the next. }
if not errorflag then begin
    page(output);
    PrintHeading;
    writeln(output,
        ' Head Body Notes ',
        ':SigCharLimit,
        ' Calls made to');
    for pretty := 1 to (SigCharLimit+37) do
        write(output, '-');
    writeln(output);
    writeln(output);
    PrintTree(superroot);
end;
end { Referencer }.