```
0001:  program Referencer(input,output);  {$D2,S+ }
0002:  {-------------------------------------------------------------------|
0003:  |                                                                   |
0004:  |                  PASCAL PROCEDURAL CROSS-REFERENCER               |
0005:  |                                                                   |
0006:  |        (c) Copyright 1979 A.H.J.Sale, Southampton, England.       |
0007:  |                                                                   |
0008:  |   DEVELOPMENT                                                     |
0009:  |   This program is a software tool developed  from  a  prototype  by  |
0010:  |   A.J.Currie at the University of Southampton, England.  The proto-  |
0011:  |   type of 231 lines of source text was used firstly as a basis  for  |
0012:  |   extensions,   and   then   rewritten  to  assure  correctness  by  |
0013:  |   A.H.J.Sale, on leave from the University  of  Tasmania  and  then  |
0014:  |   also  at  the University of Southampton.  The current version was  |
0015:  |   stabilized at 1979 December 4; the  development  time  being  es-  |
0016:  |   timated at 4 man-days from prototype to production.             |
0017:  |                                                                   |
0018:  |   PURPOSE                                                         |
0019:  |   The program reads Pascal source programs and produces two  tables  |
0020:  |   as  output.  These tables are procedural documentation and cross-  |
0021:  |   references.  One documents all procedure or function headings  in  |
0022:  |   a  format  that  illustrates  lexical  nesting.  The other tables  |
0023:  |   gives the locations of heading, block, and  body  for  each  pro-  |
0024:  |   cedure and function, and what procedures and functions it immedi-  |
0025:  |   ately calls.                                                    |
0026:  |                                                                   |
0027:  |   There is a User Manual for this program; if it has not been  pro-  |
0028:  |   vided with your installation write to:                          |
0029:  |             Department of Information Science                     |
0030:  |             University of Tasmania                                |
0031:  |             P.O.Box 252C, G.P.O. Hobart                           |
0032:  |             Tasmania  7001                                        |
0033:  |   and ask for the Technical Report on Referencer, if  it  is  still  |
0034:  |   available.  The program is written to be portable and is believed  |
0035:  |   to be in Standard Pascal.                                       |
0036:  |                                                                   |
0037:  |   Permission is granted to copy this program, store it in a comput-  |
0038:  |   er  system,  and distribute it, provided that this header comment  |
0039:  |   is retained in all copies.                                      |
0040:  |                                                                   |
0041:  |-------------------------------------------------------------------}
0042:
0043:  {-------------------------------------------------------------------|
0044:  |                                                                   |
0045:  |                          PROGRAM ASSERTIONS                       |
0046:  |                                                                   |
0047:  |   Pre-Assertion P1:                                               |
0048:  |         "The file input contains  a  representation  of  a  correct  |
0049:  |         Standard Pascal program, in the ISO Reference form."      |
0050:  |                                                                   |
0051:  |   Post-assertion P2:                                              |
0052:  |         P1 and "the file output contains a  representation  of  the  |
0053:  |         two  tables described above, which correctly describe facts  |
0054:  |         about the program."                                       |
0055:  |                                                                   |
0056:  |-------------------------------------------------------------------}
0057:
0058:  const
0059:      { This constant is the number of significant characters kept in
0060:      the identifier entries.  It can readily be changed.  It is not
0061:      advised that it be reduced below 10 (reserved words get to 9). }
0062:      SigCharLimit = 16;
0063:
0064:      { This must always be (SigCharLimit - 1).  It is used simply to
0065:      reduce the set range to have a lower bound of 0, not 1. }
```

```
0066:      SetLimit = 15;
0067:
0068:      { This constant is used to convert upper-case letters to lower-case
0069:      and vice-versa.  It should be equal to ord('a') - ord('A'). }
0070:      UCLCdisplacement = 32;
0071:
0072:      { This constant determines the size of the input line buffer.
0073:      The maximum acceptable input line is one smaller because a sentinel
0074:      space is appended to every line. }
0075:      LineLimit = 200;
0076:
0077:      { This constant determines the maximum width of the printing of the
0078:      second cross-reference table.  The program deduces how many names
0079:      will fit on a line. }
0080:      LineWidth = 132;
0081:
0082:      { This determines the indentation of the lex-levels. }
0083:      Indentation = 4;
0084:
0085:      { These constants are used for the sketchy syntax analysis.
0086:      They are collected here so that their lengths may be altered if
0087:      SigCharLimit is altered. }
0088:      Sprogram      = 'program         ';
0089:      Sprocedure    = 'procedure       ';
0090:      Sfunction     = 'function        ';
0091:      Slabel        = 'label           ';
0092:      Sconst        = 'const           ';
0093:      Stype         = 'type            ';
0094:      Svar          = 'var             ';
0095:      Sbegin        = 'begin           ';
0096:      Scase         = 'case            ';
0097:      Send          = 'end             ';
0098:      Sforward      = 'forward         ';
0099:      Spaces        = '                ';
0100:
0101:  type
0102:      Natural        = 0..maxint;
0103:      Positive       = 1..maxint;
0104:
0105:      SixChars       = packed array[1..6] of char;
0106:
0107:      SigCharRange   = 1..SigCharLimit;
0108:      SetRange       = 0..SetLimit;
0109:
0110:      PseudoString   = packed array [SigCharRange] of char;
0111:      StringCases    = set of SetRange;
0112:
0113:      LineSize       = 1..LineLimit;
0114:      LineIndex      = 0..LineLimit;
0115:
0116:      SetOfChar      = set of char;
0117:
0118:      ProcKind       = (FwdHalf,AllFwd,Shortform,Formal,Outside,NotProc);
0119:
0120:      PtrToEntry     = ^ Entry;
0121:
0122:      ListOfUsages   = ^ UsageCell;
0123:
0124:      PtrToStackCell = ^ StackCell;
0125:
0126:      TokenType      = (OtherSy,NameSy,LParenSy,RParenSy,ColonSy,
0127:                        SemiColSy,PeriodSy,AssignSy,SubRangeSy);
0128:
0129:      { This type represents a procedure or function identifier found
0130:      during processing of a program.  The fields are used as follows:
```

```
0131:        - procname & caseset = representation of name
0132:        - linenumber         = where heading starts
0133:        - startofbody        = where begin of statement-part starts
0134:        - forwardblock       = where forward-declared block starts
0135:        - status             = kind or status of name
0136:        - left,right         = subtrees of the scope-level tree
0137:        - before, after      = subtrees of the supertree
0138:        - calls              = a list of the procedures this calls
0139:        - localtree          = the scope tree for the interior
0140:      }
0141:      Entry =
0142:           record
0143:               procname     : PseudoString;
0144:               caseset      : StringCases;
0145:               linenumber   : Natural;
0146:               startofbody  : Natural;
0147:               left,right   : PtrToEntry;
0148:               before,after : PtrToEntry;
0149:               calls        : ListOfUsages;
0150:               localtree    : PtrToEntry;
0151:               case status : ProcKind of
0152:                   FwdHalf,Shortform,Formal,Outside,NotProc:
0153:                       ();
0154:                   AllFwd:
0155:                       ( forwardblock: Natural )
0156:           end;
0157:
0158:      { This type records an instance of an activation of a procedure or
0159:      function.  The next pointers maintain an alphabetically ordered
0160:      list; the what pointer points to the name of the activated code. }
0161:      UsageCell =
0162:           record
0163:               what : PtrToEntry;
0164:               next : ListOfUsages
0165:           end;
0166:
0167:      { This type is used to construct a stack which holds the current
0168:      lexical level information. }
0169:      StackCell =
0170:           record
0171:               current   : PtrToEntry;
0172:               scopetree : PtrToEntry;
0173:               substack  : PtrToStackCell
0174:           end;
0175:
0176:  var
0177:      lineno       : Natural;
0178:      chno         : LineIndex;
0179:      total        : LineIndex;
0180:      depth        : Natural;
0181:      level        : -1..maxint;
0182:      pretty       : Natural;
0183:
0184:      { These are used to align the lines of a heading. }
0185:      adjustment   : (First,Other);
0186:      movement     : integer;
0187:
0188:      { These are true, respectively, if line-buffers need to be
0189:      printed before disposal, and if any errors have occurred. }
0190:      printflag    : boolean;
0191:      errorflag    : boolean;
0192:
0193:      ch           : char;
0194:
0195:      token        : tokentype;
```

```
0196:
0197:       symbol       : PseudoString;
0198:       symbolcase   : StringCases;
0199:
0200:       savesymbol   : PseudoString;
0201:
0202:       line         : array[LineSize] of char;
0203:
0204:       superroot    : PtrToEntry;
0205:
0206:       stack        : PtrToStackCell;
0207:
0208:       { The remaining variables are pseudo-constants. }
0209:       alphabet     : SetOfChar;
0210:       alphanums    : SetOfChar;
0211:       uppercase    : SetOfChar;
0212:       digits       : SetOfChar;
0213:       usefulchars  : SetOfChar;
0214:
0215:       namesperline : Positive;
0216:
0217:       procedure PrintLine;
0218:       var
0219:           i : LineSize;
0220:       begin
0221:           write(output, lineno:5, '    ');
0222:           i := 1;
0223:           { Is this the first time in a run or not? }
0224:           if adjustment = First then begin
0225:               { Ignore any leading spaces there happen to be. }
0226:               while (i < total) and (line[i] = ' ') do
0227:                   i := succ(i);
0228:               { Compute the adjustment needed for other lines. }
0229:               movement := (level * Indentation) - (i - 1);
0230:               adjustment := Other;
0231:               { Insert any necessary indentation }
0232:               if level > 0 then
0233:                   write(output, ' ': (level*Indentation))
0234:           end else begin
0235:               { It wasn't the first time, so try to adjust this
0236:               line to align with its mother. }
0237:               if movement > 0 then begin
0238:                   write(output, ' ':movement)
0239:               end else if movement < 0 then begin
0240:                   while (i < total) and (line[i] = ' ') and
0241:                       (i <= - movement) do begin
0242:                       i := succ(i)
0243:                   end
0244:               end
0245:           end;
0246:           { Write out the line. }
0247:           while i < total do begin
0248:               write(output, line[i]);
0249:               i := succ(i)
0250:           end;
0251:           writeln(output)
0252:       end; { PrintLine }
0253:
0254:       procedure Error(e: Positive);
0255:       { This procedure is the error message repository. }
0256:       begin
0257:           errorflag := true;
0258:           write(output, 'FATAL ERROR - ');
0259:           case e of
0260:            1: write(output, 'No "program" word');
```

```
0261:           2: write(output, 'No identifier after prog/proc/func');
0262:           3: write(output, 'Token after heading expected');
0263:           4: write(output, 'Lost ".", check begin/case/ends');
0264:           5: write(output, 'Same name, but not forward-declared')
0265:         end;
0266:         { We shall print the offending line too. }
0267:         writeln(output, ' - AT FOLLOWING LINE');
0268:         adjustment := first;
0269:         PrintLine
0270:     end; { Error }
0271:
0272:     procedure NextCh;
0273:     begin
0274:         if chno = total then begin
0275:             if printflag then
0276:                 PrintLine;
0277:             total := 0;
0278:             while not eoln(input) do begin
0279:                 total := succ(total);
0280:                  read(input, line[total])
0281:             end;
0282:             total := succ(total);
0283:             line[total] := ' ';
0284:             readln(input);
0285:             lineno := lineno + 1;
0286:             chno := 1;
0287:             ch := line[1]
0288:         end else begin
0289:             chno := succ(chno);
0290:             ch := line[chno]
0291:         end
0292:     end; { NextCh }
0293:
0294:     procedure Push(newscope: PtrToEntry);
0295:     var
0296:         newlevel: PtrToStackCell;
0297:     begin
0298:         new(newlevel);
0299:         newlevel^.current := newscope;
0300:         newlevel^.scopetree := nil;
0301:         newlevel^.substack := stack;
0302:         stack := newlevel;
0303:         level := level + 1
0304:     end; { Push }
0305:
0306:     procedure Pop;
0307:     var
0308:         oldcell: PtrToStackCell;
0309:     begin
0310:         stack^.current^.localtree := stack^.scopetree;
0311:         oldcell := stack;
0312:         stack := oldcell^.substack;
0313:         { *** dispose(oldcell); *** }
0314:         level := level - 1
0315:     end; { Pop }
0316:
0317:     procedure FindNode(var match   : Boolean;
0318:                        var follow  : PtrToEntry;
0319:                            thisnode: PtrToEntry);
0320:     begin
0321:         match := false;
0322:         while (thisnode <> nil) and not match do begin
0323:             follow := thisnode;
0324:             if savesymbol < thisnode^.procname then
0325:                 thisnode := thisnode^.left
```

```
0326:              else if savesymbol > thisnode^.procname then
0327:                  thisnode := thisnode^.right
0328:              else
0329:                  match := true
0330:          end
0331:      end; { FindNode }
0332:
0333:      function MakeEntry (mainprog: Boolean;
0334:                          proc    : Boolean): PtrToEntry;
0335:      { The first parameter is true if the name in symbol is the
0336:      program identifier, which has no scope.  The second parameter
0337:      is true if the name in symbol is that of a procedure or function.
0338:      The result returned is the identification of the relevant record. }
0339:      var
0340:          newentry, node: PtrToEntry;
0341:          located: Boolean;
0342:
0343:          procedure PutToSuperTree(newnode: PtrToEntry);
0344:          { This procedure takes the entry that has been created by
0345:          MakeEntry and inserted into the local tree, and also links
0346:          it into the supertree. }
0347:          var
0348:              place: PtrToEntry;
0349:
0350:              procedure FindLeaf;
0351:              { FindLeaf searches the supertree to find where this
0352:              node should be placed.  It will be appended to a leaf
0353:              of course, and placed after entries with the same
0354:              name. }
0355:              var
0356:                  subroot : PtrToEntry;
0357:              begin
0358:                  subroot := superroot;
0359:                  while subroot <> nil do begin
0360:                      place := subroot;
0361:                      if savesymbol < subroot^.procname then
0362:                          subroot := subroot^.before
0363:                      else
0364:                          subroot := subroot^.after
0365:                  end
0366:              end; { FindLeaf }
0367:
0368:          begin { PutToSuperTree }
0369:              if superroot = nil then begin
0370:                  { Nothing in the supertree yet. }
0371:                  superroot := newnode
0372:              end else begin
0373:                  { Seek the right place }
0374:                  FindLeaf;
0375:                  with place^ do begin
0376:                      if savesymbol < procname then
0377:                          before := newnode
0378:                      else
0379:                          after := newnode
0380:                  end
0381:              end
0382:          end; { PutToSuperTree }
0383:
0384:      begin { MakeEntry }
0385:          located := false;
0386:          savesymbol := symbol;
0387:          if mainprog then begin
0388:              new(newentry);
0389:          end else if stack^.scopetree = nil then begin
0390:              { Nothing here yet. }
```

```
0391:             new(newentry);
0392:             stack^.scopetree := newentry
0393:         end else begin
0394:             { Seek the identifier in the tree. }
0395:             FindNode(located, node, stack^.scopetree);
0396:             if not located then begin
0397:                 { Normal case, make an entry. }
0398:                 new(newentry);
0399:                 with node^ do
0400:                     if symbol < procname then
0401:                         left := newentry
0402:                     else
0403:                         right := newentry
0404:             end
0405:         end;
0406:         if not located then begin
0407:             { Here we initialize all the fields }
0408:             with newentry^ do begin
0409:                 procname := symbol;
0410:                 caseset := symbolcase;
0411:                 linenumber := lineno;
0412:                 startofbody := 0;
0413:                 if proc then
0414:                     status := Shortform
0415:                 else
0416:                     status := NotProc;
0417:                 left := nil;
0418:                 right := nil;
0419:                 before := nil;
0420:                 after := nil;
0421:                 calls := nil;
0422:                 localtree := nil
0423:             end;
0424:             MakeEntry := newentry;
0425:             if proc then begin
0426:                 PutToSuperTree(newentry);
0427:                 Push(newentry)
0428:             end
0429:         end else begin
0430:             { Well, It'd better be forward or else. }
0431:             MakeEntry := node;
0432:             Push(node);
0433:             if node^.status = FwdHalf then begin
0434:                 stack^.scopetree := node^.localtree;
0435:                 node^.status := AllFwd;
0436:                 node^.forwardblock := lineno
0437:             end else begin
0438:                 Error(5)
0439:             end
0440:         end
0441:     end; { MakeEntry }
0442:
0443:     procedure PrintTree(root: PtrToEntry);
0444:     var
0445:         thiscell: ListOfUsages;
0446:         count: Natural;
0447:
0448:         procedure ConditionalWrite(n: Natural;
0449:                           substitute: SixChars);
0450:         begin
0451:             { Write either the substitute string or a number. }
0452:             if n = 0 then
0453:                 write(output, substitute)
0454:             else
0455:                 write(output, n:6)
```

```
0456:          end; { ConditionalWrite }
0457:
0458:          procedure NameWrite(p : PtrToEntry);
0459:          var
0460:              s : SetRange;
0461:          begin
0462:              for s := 0 to SetLimit do begin
0463:                  if s in p^.caseset then
0464:                      write(output,
0465:                          chr(ord(p^.procname[s+1])-UCLCdisplacement))
0466:                  else
0467:                      write(output, p^.procname[s+1])
0468:              end
0469:          end; { NameWrite }
0470:
0471:      begin { PrintTree }
0472:          if root <> nil then
0473:              with root^ do begin
0474:                  PrintTree(before);
0475:
0476:                  writeln(output);
0477:                  write(output, linenumber: 5);
0478:                  ConditionalWrite(startofbody, '      ');
0479:                  case status of
0480:                      FwdHalf,NotProc:
0481:                          write(output, '   eh?');
0482:                      Formal:
0483:                          write(output, '   fml');
0484:                      Outside:
0485:                          write(output, '   ext');
0486:                      Shortform:
0487:                          write(output, '      ');
0488:                      AllFwd:
0489:                          write(output, forwardblock:6)
0490:                  end;
0491:                  write(output, '  ');
0492:                  NameWrite(root);
0493:                  write(output, ' :');
0494:                  thiscell := calls;
0495:                  count := 0;
0496:                  while thiscell <> nil do begin
0497:                      if ((count mod namesperline) = 0) and (count <> 0)
0498:                          then begin
0499:                          writeln(output);
0500:                          write(output, ' ':35, ' :')
0501:                      end;
0502:                      write(output, ' ');
0503:                      NameWrite(thiscell^.what);
0504:                      thiscell := thiscell^.next;
0505:                      count := count + 1
0506:                  end;
0507:                  writeln(output);
0508:
0509:                  PrintTree(after)
0510:              end
0511:      end; { PrintTree }
0512:
0513:      procedure NextToken;
0514:      { This procedure produces the next "token" in a small set of
0515:      recognized tokens.  Most of these serve an incidental purpose;
0516:      the prime purpose is to recognize names (res'd words or identifiers).
0517:      It serves also to skip dangerous characters in comments, strings,
0518:      and numbers. }
0519:
0520:          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 comsume 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:                 while ch in digits do
0558:                     NextCh
0559:             end
0560:         end; { IgnoreNumbers }
0561:
0562:         procedure ReadIdent;
0563:         { This procedure reads in an identifier }
0564:         var
0565:             j : Positive;
0566:         begin
0567:             token := NameSy;
0568:             symbol := Spaces;
0569:             symbolcase := [];
0570:             j := 1;
0571:             while (j <= SigCharLimit) and (ch in alphanums) do begin
0572:                 if ch in uppercase then begin
0573:                     symbol[j] := chr(ord(ch) + UCLCdisplacement);
0574:                     symbolcase := symbolcase + [j-1]
0575:                 end else begin
0576:                     symbol[j] := ch
0577:                 end;
0578:                 j := j+1;
0579:                 NextCh
0580:             end;
0581:             { In case there is a tail, skip it. }
0582:             while ch in alphanums do
0583:                 NextCh
0584:         end; { ReadIdent }
0585:
```

```
0586:      begin { NextToken }
0587:          token := OtherSy;
0588:          repeat
0589:              if ch in usefulchars then begin
0590:                  case ch of
0591:
0592:                   ')': begin
0593:                            NextCh;
0594:                            token := RParenSy
0595:                        end;
0596:
0597:                   '(': begin
0598:                            NextCh;
0599:                            if ch = '*' then begin
0600:                                IgnoreComment
0601:                            end else begin
0602:                                token := LParenSy
0603:                            end
0604:                        end;
0605:
0606:                   '{': begin
0607:                            IgnoreComment
0608:                        end;
0609:
0610:                  '''': begin
0611:                            NextCh;
0612:                            while ch <> '''' do
0613:                                NextCh;
0614:                            NextCh
0615:                        end;
0616:
0617:                   '0','1','2','3','4','5','6','7','8','9':
0618:                        begin
0619:                            IgnoreNumbers
0620:                        end;
0621:
0622:                   ':': begin
0623:                            NextCh;
0624:                            if ch = '=' then begin
0625:                                token := AssignSy;
0626:                                NextCh
0627:                            end else begin
0628:                                token := ColonSy
0629:                            end
0630:                        end;
0631:
0632:                   '.': begin
0633:                            NextCh;
0634:                            if ch <> '.' then
0635:                                token := PeriodSy
0636:                            else begin
0637:                                token := SubRangeSy;
0638:                                NextCh
0639:                            end
0640:                        end;
0641:
0642:                   ';': begin
0643:                            NextCh;
0644:                            token := SemiColSy
0645:                        end;
0646:
0647:                   'A','B','C','D','E','F','G','H','I','J','K','L','M',
0648:                   'N','O','P','Q','R','S','T','U','V','W','X','Y','Z',
0649:                   'a','b','c','d','e','f','g','h','i','j','k','l','m',
0650:                   'n','o','p','q','r','s','t','u','v','w','x','y','z':
```

```
0651:                       begin
0652:                           ReadIdent
0653:                       end
0654:
0655:                   end
0656:               end else begin
0657:                   { Uninteresting character }
0658:                   NextCh
0659:               end
0660:           until token <> OtherSy
0661:       end; { NextToken }
0662:
0663:       procedure ProcessUnit(programid: Boolean);
0664:       { This procedure processes a program unit.  It is called on
0665:       recognition of its leading token = program/procedure/function.
0666:       The parameter records whether we currently have the main program
0667:       identifier in the token, or not.  It doesn't have scope. }
0668:       var
0669:           at : PtrToEntry;
0670:
0671:           function NameIsInScope: Boolean;
0672:           { This function is called during the declaration phase
0673:           of a block, and has to find any procedure which gets
0674:           renamed by the scope rules. }
0675:           var
0676:               llevel        : PtrToStackCell;
0677:               discovered    : Boolean;
0678:               where         : PtrToEntry;
0679:           begin
0680:               llevel := stack;
0681:               discovered := false;
0682:               savesymbol := symbol;
0683:               while (llevel <> nil) and not discovered do begin
0684:                   FindNode(discovered, where, llevel^.scopetree);
0685:                   if not discovered then
0686:                       llevel := llevel^.substack
0687:               end;
0688:               if discovered then
0689:                   NameIsInScope := (where^.status <> NotProc)
0690:               else
0691:                   NameIsInScope := false
0692:           end; { NameIsInScope }
0693:
0694:           procedure ProcessBlock;
0695:           { This procedure is called by ProcessUnit when it has recognized
0696:           the start of a block.  It handles the processing of the block. }
0697:           var
0698:               address: PtrToEntry;
0699:
0700:               procedure CrossReferencer;
0701:               { CrossReferencer is called whenever we have a name which
0702:               might be a call to a procedure or function.  The only way
0703:               we tell is by looking in the table to see.  If it is, then
0704:               the list of usages of the procedure we are in is scanned and
0705:               possibly extended. }
0706:               var
0707:                   newcell : ListOfUsages;
0708:                   ptr     : ListOfUsages;
0709:                   home    : PtrToEntry;
0710:                   slevel  : PtrToStackCell;
0711:                   found   : Boolean;
0712:
0713:                   procedure FindCell;
0714:                   { FindCell is used to scan a List Of Usages to determine
0715:                   whether the name already appears there.  If not, it
```

```
0716:                 leaves ptr pointing to the tail of the list so that an
0717:                 addition can be made. }
0718:                 var
0719:                     nextptr  : ListOfUsages;
0720:                 begin
0721:                     found := false;
0722:                     nextptr := stack^.current^.calls;
0723:                     if nextptr <> nil then
0724:                         repeat
0725:                             ptr := nextptr;
0726:                             found := (ptr^.what^.procname = savesymbol);
0727:                             nextptr := ptr^.next
0728:                         until found or (nextptr = nil)
0729:                     else
0730:                         ptr := nil
0731:                 end; { FindCell }
0732:
0733:             begin { CrossReferencer }
0734:                 slevel := stack;
0735:                 found := false;
0736:                 while (slevel <> nil) and not found do begin
0737:                     FindNode(found, home, slevel^.scopetree);
0738:                     if not found then
0739:                         slevel := slevel^.substack
0740:                 end;
0741:                 if found then begin
0742:                     if home^.status <> NotProc then begin
0743:                         FindCell;
0744:                         if not found then begin
0745:                             new(newcell);
0746:                             if ptr <> nil then
0747:                                 ptr^.next := newcell
0748:                             else
0749:                                 stack^.current^.calls := newcell;
0750:                             newcell^.what := home;
0751:                             newcell^.next := nil
0752:                         end
0753:                     end
0754:                 end
0755:             end; { CrossReferencer }
0756:
0757:             procedure ScanForName;
0758:             { This procedure is required to go forward until the
0759:             current token is a name (reserved word or identifier). }
0760:             begin
0761:                 NextToken;
0762:                 while token <> NameSy do
0763:                     NextToken
0764:             end; { ScanForName }
0765:
0766:         begin { ProcessBlock }
0767:             while (symbol <> Sbegin) do begin
0768:                 while (symbol <> Sbegin) and (symbol <> Sprocedure) and
0769:                     (symbol <> Sfunction) do begin
0770:                     ScanForName;
0771:                     if NameIsInScope then begin
0772:                         address := MakeEntry(false, false);
0773:                         { MakeEntry made its status NotProc }
0774:                     end
0775:                 end;
0776:                 if symbol <> Sbegin then begin
0777:                     ProcessUnit(false);
0778:                     ScanForName
0779:                 end
0780:             end;
```

```
0781:               { We have now arrived at the body }
0782:               depth := 1;
0783:               stack^.current^.startofbody := lineno;
0784:               NextToken;
0785:               while depth <> 0 do begin
0786:                   if token <> NameSy then begin
0787:                       NextToken
0788:                   end else begin
0789:                       if (symbol = Sbegin) or (symbol = Scase) then begin
0790:                           depth := depth + 1;
0791:                           NextToken
0792:                       end else if (symbol = Send) then begin
0793:                           depth := depth - 1;
0794:                           NextToken
0795:                       end else begin
0796:                           { This name is a candidate call.  But first we
0797:                           must eliminate assignments to function values. }
0798:                           savesymbol := symbol;
0799:                           NextToken;
0800:                           if token <> AssignSy then begin
0801:                               CrossReferencer
0802:                           end else begin
0803:                               NextToken
0804:                           end
0805:                       end
0806:                   end
0807:               end
0808:           end; { ProcessBlock }
0809:
0810:           procedure ScanParameters;
0811:           { This procedure scans the parameter list because at the outer
0812:           level there may be a formal procedure we ought to know about. }
0813:           var
0814:               which : PtrToEntry;
0815:
0816:               procedure ScanTillClose;
0817:               { This procedure is called when a left parenthesis is
0818:               detected, and its task is to find the matching right
0819:               parenthesis.  It does this recursively. }
0820:               begin
0821:                   NextToken;
0822:                   while token <> RParenSy do begin
0823:                       if token = LParenSy then
0824:                           ScanTillClose;
0825:                       NextToken
0826:                   end
0827:               end; { ScanTillClose }
0828:
0829:           begin { ScanParameters }
0830:               NextToken;
0831:               while token <> RParenSy do begin
0832:                   if (token = NameSy) then begin
0833:                       if (symbol = Sprocedure) or
0834:                           (symbol = Sfunction) then begin
0835:                           { A formal procedural/functional parameter. }
0836:                           NextToken;
0837:                           if token = NameSy then begin
0838:                               which := MakeEntry(false, true);
0839:                               which^.status := Formal;
0840:                               Pop;
0841:                               NextToken;
0842:                               if token = LParenSy then begin
0843:                                   { Skip interior lists. }
0844:                                   ScanTillClose
0845:                               end
```

```
0846:                           end else begin
0847:                               Error(2);
0848:                               NextToken
0849:                           end
0850:                       end else begin
0851:                           if NameIsInScope then
0852:                               which := MakeEntry(false, false);
0853:                           NextToken
0854:                       end
0855:                   end else begin
0856:                       NextToken
0857:                   end
0858:               end;
0859:               NextToken
0860:           end; { ScanParameters }
0861:
0862:       begin { ProcessUnit }
0863:           printflag := true;
0864:           adjustment := First;
0865:           NextToken;
0866:           if token <> NameSy then
0867:               Error(2)
0868:           else begin
0869:               { We now have the name to store away. }
0870:               at := MakeEntry(programid, true);
0871:               while not (token in [LParenSy,SemiColSy,ColonSy]) do
0872:                   NextToken;
0873:               if token = LParenSy then
0874:                   ScanParameters;
0875:               while token <> SemiColSy do
0876:                   NextToken;
0877:               PrintLine;
0878:               { We have now printed the procedure heading. }
0879:               printflag := false;
0880:               writeln(output);
0881:               { Our next task is to see if there is an attached block. }
0882:               NextToken;
0883:               if token <> NameSy then
0884:                   Error(3)
0885:               else begin
0886:                   if (symbol <> Slabel) and (symbol <> Sconst) and
0887:                       (symbol <> Stype) and (symbol <> Sprocedure) and
0888:                       (symbol <> Sfunction) and (symbol <> Svar) and
0889:                       (symbol <> Sbegin) then begin
0890:                       { Bloody directive, mate. }
0891:                       if symbol = Sforward then
0892:                           at^.status := FwdHalf
0893:                       else
0894:                           at^.status := Outside;
0895:                       Pop
0896:                   end else begin
0897:                       ProcessBlock;
0898:                       Pop
0899:                   end
0900:               end
0901:           end
0902:       end; { ProcessUnit }
0903:
0904:   { *** ------------------------------------------------------|
0905:   |                                                           |
0906:   |   This procedure outlines what is needed  to insert the   |
0907:   |   predefined names into Referencer's tables.  De-box it   |
0908:   |   and extend it as needed.                                |
0909:   |                                                           |
0910:   |   procedure BuildPreDefined;                              |
```

```
0911:  |   const                                                  |
0912:  |       NoOfNames = 2;                                    |
0913:  |   type                                                   |
0914:  |       NamesIndex   = 1..NoOfNames;                      |
0915:  |   var                                                    |
0916:  |       kk   : NamesIndex;                                |
0917:  |       tt   : array[NamesIndex] of PseudoString;          |
0918:  |       hohum: PtrToEntry;                                |
0919:  |   begin                                                  |
0920:  |       tt[01] := 'new            ';                      |
0921:  |       tt[02] := 'writeln        ';                      |
0922:  |       caseset := [];                                     |
0923:  |       for kk := 1 to NoOfNames do begin                  |
0924:  |           symbol := tt[kk];                              |
0925:  |           hohum := MakeEntry(false,false);               |
0926:  |           hohum^.status := Outside;                      |
0927:  |       end;                                               |
0928:  |   end;                                                   |
0929:  |                                                          |
0930:  |--------------------------------------------------- *** }
0931:
0932:      procedure PrintHeading;
0933:      begin
0934:          writeln(output, 'Procedural Cross-Referencer - Version S-02.02');
0935:          writeln(output, '=========================================');
0936:          writeln(output)
0937:      end; { PrintHeading }
0938:
0939:  begin { Referencer }
0940:      superroot := nil;
0941:      { Here we construct an outer-scope stack entry.  This is needed
0942:      to hold any pre-defined names.  The distributed version does not
0943:      include any of these, but they are easily provided.  See the
0944:      outlines in the code marked with *** if you want this feature. }
0945:      new(stack);
0946:      with stack^ do begin
0947:          current := nil;
0948:          scopetree := nil;
0949:          substack := nil
0950:      end;
0951:
0952:      printflag := false;
0953:
0954:      uppercase := ['A','B','C','D','E','F','G','H','I','J','K','L','M',
0955:                    'N','O','P','Q','R','S','T','U','V','W','X','Y','Z'];
0956:      alphabet  := uppercase +
0957:                   ['a','b','c','d','e','f','g','h','i','j','k','l','m',
0958:                    'n','o','p','q','r','s','t','u','v','w','x','y','z'];
0959:      digits    := ['0','1','2','3','4','5','6','7','8','9'];
0960:      alphanums := alphabet + digits { *** + ['_'] *** } ;
0961:      usefulchars := alphabet + digits +
0962:                   ['(', ')', '{', '.', ':', ';', ''''];
0963:
0964:      namesperline := (LineWidth - (SigCharLimit + 21)) div
0965:          (SigCharLimit + 1);
0966:
0967:      { *** If you want to introduce some options, this is the place
0968:      to insert the call to your OptionAnalyser.  None is provided
0969:      with the standard tool because the requirements vary widely
0970:      across user environments.  The probable options that might be
0971:      provided are (a) whether pre-declared names should appear in
0972:      the call lists, (b) how many columns are to be printed in them
0973:      (namesperline), (c) whether underscore is permitted in identifiers,
0974:      and perhaps whether output should be completely in upper-case
0975:      letters.  The first option (a) requires a call to BuildPreDefined
```

```
0976:      just below this point, after analysing options... }
0977:
0978:      total := 0;
0979:      chno := 0;
0980:      lineno := 0;
0981:      level := -1;
0982:      errorflag := false;
0983:      { *** BuildPreDefined; *** }
0984:
0985:      { *** } page(output);  { *** }
0986:      PrintHeading;
0987:      writeln(output, ' Line   Program/procedure/function heading');
0988:      for pretty := 1 to 43 do
0989:          write(output, '-');
0990:      writeln(output);
0991:      writeln(output);
0992:      { Now we need to get the first token, which should be program. }
0993:      ch := ' ';  { *** bug fix - JTE SSRFC U of Minn  1981-03-18 }
0994:      NextToken;
0995:      if token <> NameSy then
0996:          Error(1)
0997:      else if symbol <> Sprogram then
0998:          Error(1)
0999:      else begin
1000:          ProcessUnit(true);
1001:          { Having returned, there ought to be a period here. }
1002:          if not errorflag then begin
1003:              { We check all tokens that begin with a period because
1004:              what occurs after the closing period is nothing to do
1005:              with us. }
1006:              if (token <> PeriodSy) and (token <> SubRangeSy) then
1007:                  Error(4)
1008:              else begin
1009:                  adjustment := First;
1010:                  PrintLine
1011:              end
1012:          end
1013:      end;
1014:      { Completed Phase One - now for the next. }
1015:      if not errorflag then begin
1016:          page(output);
1017:          PrintHeading;
1018:          writeln(output,
1019:              ' Head  Body  Notes ',
1020:              ' ':SigCharLimit,
1021:              '   Calls made to');
1022:          for pretty := 1 to (SigCharLimit+37) do
1023:              write(output, '-');
1024:          writeln(output);
1025:          PrintTree(superroot);
1026:          writeln(output)
1027:      end
1028:  end { Referencer }.

THAT'S ALL FOLKS!      LINES: 1028   CHARACTERS: 37544
```