A PROCEDURE HIERARCHY GENERATOR FOR PASCAL

by

KENNETH D. HARMON
B. A., Pittsburg State University, 1967

A MASTER'S REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
MANHATTAN, KANSAS

1980

Approved by:
# TABLE OF CONTENTS

I. INTRODUCTION ................................................................................. 1

II. USER INFORMATION ................................................................. 7
    Executing the Generator ......................................................... 7
    Reading and Understanding the Generator Output ............... 7

III. SYSTEM SPECIFICATIONS ......................................................... 13
    System Overview ................................................................. 13
    Software Structure ............................................................. 13
    Output Design .................................................................. 17
    Data Structure .................................................................. 17
    Routines ........................................................................... 20
    Error Processing ............................................................... 25
    Future Extensions ............................................................. 28

IV. BIBLIOGRAPHY ........................................................................ 31

V. APPENDICES ............................................................................

    APPENDIX A - Procedure Hierarchy Generator .... A-1
        Output Listing
    APPENDIX B - Output Syntax Diagram .......... B-1
    APPENDIX C - Output Specifications .......... C-1
    APPENDIX D - Implementation Data Structure .... D-1
    APPENDIX E - Procedure Hierarchy Generator .... E-1
        Source Listing
FIGURES AND TABLES

Figure I.1 - Sample Non-Graphical Display ............... 2
Figure III.1 - Generator Program Structure ............... 14
Figure III.2 - Generator Routines Structure ............... 15
Figure III.3 - Build Hash Table Routines Structure ...... 16
   (Input Routines)
Figure III.4 - Print Hash Table Routines Structure ...... 16
   (Output Routines)
Figure III.5 - Structure of the Build Hash Table ...... 20
   Routines
Figure III.6 - Structure of the Print Hash Table ...... 24
   Routines

Table III.1 - Generator Output Formats ................. 17
I. INTRODUCTION

This Master's Report presents a software engineering tool for the generation of the hierarchies of procedures in sequential PASCAL computer programs.

This tool is a method to non-graphically represent the structure of PASCAL programs as pertains to their procedure calling sequence. Consider a program in which the main routine calls two procedures and each of these two called procedures calls one common subprocedure and one other subprocedure. (NOTE. Hereinafter, reference to procedures which are called by other procedures will be termed subprocedures.) The structure of this program can be depicted by several methods. The method implemented by this tool is one which is relatively easy to automate and one which conveys essentially as much information to the user as does other methods, e.g., access graphs. Applying this method to the sample program described above, the program structure would appear as in Figure I.1. Note that the display depicts the calling structure, not the physical order of the procedure declarations. With the addition of other pieces of information to the display, as described later in this report, this software engineering tool will
greatly assist users in understanding the structure of sequential PASCAL programs.

```
MAIN
    PROC3
    PROC1
    PROC2
    PROC5
    PROC4
    PROC2
END.
```

Figure I.1 Sample Non-Graphical Display

One of the PASCAL language design goals was that of simplicity. Programs such as compilers are so large that one cannot understand them all at once. Thus, large programs must be reasoned about in smaller pieces. Per Brinch Hansen [BRI77] states that "...it would be ideal if they (the pieces) were no more than one page of text each so that they can be comprehended at a glance". This philosophy of keeping PASCAL programs simple is achieved, primarily, through the extensive use of procedure definitions and calls. Procedures themselves can be kept small (i.e., no more than one page of text) by also invoking calls to subprocedures, rather than employing straight line code. This sequence of events then can be repeated until such time that each distinctive portion of a program has been separated into many small procedures, each of which is physically small and readily understandable.
A side effect of this technique of structuring programs, however, is the difficulty it causes individuals who are attempting to gain a general knowledge of a large program or are attempting to manually trace execution in a large program. For example, consider students or analysts studying PASS1 of the KSU PAS32 PASCAL compiler. PASS1, a 1600 line program, has on the order of 50 procedure declarations and 2500 possible calls to those procedures which are nested to a maximum of nine levels deep. As is readily apparent, an individual studying PASS1 listings would have to maintain extensive handwritten records to keep track of his current location in each procedure as he progressed deeper into and then backed out of the nested procedures. This administrative overhead detracts greatly from the substantive effort which can be applied to studying or solving an actual problem at hand.

In any discussion of computer programming, reference is inevitably made to the pattern of control of programs and to the paths which may be traversed in the course of program execution [KAR60]. To display this structure for PASCAL programs and to mitigate the difficulties discussed in the preceding paragraph, a procedure hierarchy generator was designed and implemented. The generator outputs the procedure calling sequence of programs under investigation. The concept for this generator is similar to the program graph concept. In recent years, applications of graph
theory to computer programs have given fruitful results and attracted more and more attention [PAI77]. A program graph is a graph structured model of a program exhibiting the flow relation or connection among the statements in the program. Graph theory presents a unified approach that provides insight into program structure without regard for the level of detail under the structure. Graph theory facilitates the understanding of the operation of large programs. While graph theory applies to the statement level of a program, the procedure hierarchy generator functions at the procedure level of a program. However, because of the extensive use of procedures in PASCAL, the generator provides similar benefits to the PASCAL user as program graphs provide to users of other programming languages.

Another precedent for the procedure hierarchy generator is IBM's technique for graphically displaying program structure—Hierachy plus Input-Process-Output (HIPO) [PRE76]. HIPO consists of two basic components: a hierarchy chart, which shows how each function is divided into subfunctions; and input-output-process charts, which express each function in the hierarchy in terms of its input and output. While HIPO is based on functions and not specifically on procedures, the primary purpose of both HIPO and the generator described in this report are similar—to assist the user in understanding large computer programs.
The generator assists users primarily in three major areas. Users of this software engineering tool are able to:

- Readily ascertain the structure of programs under investigation.
- Eliminate the requirement for extensive hand-written records during manual execution traces of programs under investigation.
- Additionally, use the output to verify that user programs being created are actually coded as designed (as pertains to the procedure invoking structure).

The above capabilities are provided to users via the output produced by the generator. The user’s view of this output is fully described in Part II of this report; output specifications are contained in Part III. Some of the more salient features of the generator are:

- Input programs must be free of all compilation errors to insure 100% reliability of generator output.
- Output is based on the sequential appearance of procedure calls in the input program source code, not on the logical placement of the calls. That is, the generator ignores all conditional statements, logic, etc. Hence, the output depicts sequential calls, not the actual run-time sequence of calls.
- Procedure parameters are disregarded during output formulation.
- Procedures must be explicitly called in the source program to appear in the output.
- The output is composed primarily of data lines which contain the input program line number where the called procedure was declared, the input program line number where the procedure was called, and the called procedure name. The program structure is depicted via indentation.
Testing of the generator code for logic errors was based on the use of existing KSU PASCAL programs and specially-designed PASCAL programs (written primarily to test boundary-type conditions) as input to the generator. The generator output was then hand-checked against the source code of the test programs, and errors in the generator logic were corrected as they were discovered. Existing programs utilized as test programs were the ten PAS32 compiler modules, the PEDIT program, and the graphics package.

The procedure hierarchy generator, as a stand-alone software package, adds a powerful software engineering tool to the KSU software package library. Users employing this tool can significantly increase their understanding of large PASCAL programs and, simultaneously, decrease the time and effort required to do so.
II. USER INFORMATION.

Part II is intended to serve as a user's guide to the generator and to reading and understanding the generator output. A complete working knowledge of sequential PASCAL or the INTERDATA 8/32 is not required in order to use the generator or to understand its output.

Executing the Generator

To utilize the generator, the user must have a sequential PASCAL program in his disk files. The program need not be error-free in order to be input to the generator; however, if the program is not error-free through PASS5 of the PAS32 compiler, then the results produced by the generator cannot be considered 100% reliable. Once the user is signed onto the INTERDATA 8/32, one command is all that is required to start generator execution. This command is

"GENPROC inputfilename,outputfilename"

where inputfilename is the name of the source program to be input and outputfilename is the name of the output file or device where the generator output is to be written.

Reading and Understanding the Generator Output

A. Output Data Lines.

1. General. Appendix A contains the output
created by inputting the generator source code through the
generator itself. Refer to this appendix during the
following discussion of the output data lines. Appendix A
can also be correlated with the generator source listing at
Appendix E.

2. Examine output line 1. 1178 MAIN indicates
that the main routine begins at line 1178. This program
line contains the initial "BEGIN" instruction of the main
routine.

3. Examine output lines 1, 2. 1112 1179 INITIALIZE
indicates that the main routine, at line 1179, calls
procedure INITIALIZE which has been declared at program line
1112.

4. Examine output lines 2, 3. 198...1129
WRITEMSTRING indicates that the INITIALIZE procedure, at line
1129, calls a subprocedure WRITEMSTRING which has been
declared at program line 198.

5. Examine output line 1l. 355...414 GETCHAR(8)
indicates that the GETCHAR procedure structure has been
printed previously beginning at output line 8.

6. Examine output line 16. 82...399 WRITE[2]
indicates that the procedure being expanded
(UNBALANCED_ERROR in this instance) makes two consecutive
calls to a subprocedure WRITE (no intervening calls to a
different subprocedure).

7. Examine output line 24. 355...500
GETCHAR[2](8) indicates that the procedure being expanded
(GETWORD in this instance) makes two consecutive calls to a subprocedure GETCHAR and that the structure of GETCHAR has been printed previously beginning at output line 8.

8. Examine output line 108. 1191 END. indicates that the main routine ends at line 1191. This program line contains the final "END" instruction of the input program.

B. Warning/Error Messages.

1. WARNING: IF YOUR INPUT PROGRAM HAS COMPILATION ERRORS IN PASS1 THROUGH PASS5, THE FOLLOWING OUTPUT CANNOT BE CONSIDERED RELIABLE.

   Meaning: This message is printed each time the generator processes an input program and produces output data lines. Its purpose is to alert the user to the fact that even though the generator has executed to a normal termination, the output produced may be unreliable if the input program has syntax, semantic, and/or type errors in it.

2. ***** PROGRAM CONTAINS NO PROCEDURES. *****

   Meaning: This message is printed when the generator processes an input program that does not contain any procedure declarations. No other output is produced.

3. ***** PROGRAM CONTAINS PROCEDURES; HOWEVER, ***** ***** MAIN DOES NOT INVOKE ANY OF THEM. *****

   Meaning: This message is printed when the generator processes an input program that contains procedure declarations, but the main routine does not call any of these procedures. This message will be produced even though
the procedures themselves may call other subprocedures. No other output is produced.

4. * THE PROGRAM BEING ANALYZED HAS TOO MANY * 
* PROCEDURES FOR THE GENERATOR, AS * 
* CURRENTLY SET UP, CONTACT OPERATIONS TO * 
* REQUEST THAT THIS ARBITRARY LIMIT BE * 
* RAISED. *

Meaning: This message is printed when the generator processes an input program that contains more procedure declarations than the static hash table (a hash table is used as the generator implementation data structure) has slots to store information on each declaration. No other output is produced. Excessively long procedure names (names > 20 characters) reduce the number of slots available, so the user may shorten his procedure names and resubmit his program. If this action does not eliminate the problem, then the physical size of the hash table will have to be increased by operations personnel.

5. * THE PROCEDURE BEING ANALYZED HAS TOO MANY * 
* SUBPROCEDURE CALLS FOR THE GENERATOR AS * 
* CURRENTLY SET UP, CONTACT OPERATIONS TO * 
* REQUEST THAT THIS ARBITRARY LIMIT BE * 
* RAISED. *

Meaning: This message is printed when the generator processes an input program that contains more subprocedure calls within procedures or procedure calls within the main routine than the static hash table has slots to store information on each procedure call. If this occurs within a procedure, the message is printed, and the generator resumes normal activities at the next logical point. If this occurs within the main routine, the message
is printed and no other output is produced. This situation may be remedied by having operations personnel increase the number of subprocedure calls allowed per procedure.

6. * THE PROCEDURE CURRENTLY BEING EXPANDED HAS A * * NESTING LEVEL DEEPER THAN 19. THIS EXCEEDS * * THE PHYSICAL PAPER PRINTOUT LIMITATIONS. * * THE PROCEDURE WILL NOT BE EXPANDED FURTHER. * *

Meaning: This message is printed when the generator processes an input program that contains procedure calls which are so deeply nested that when the structure of the program is output, the printing, if continued, would run off the right-hand side of the paper. This situation can arise because of the technique used to display the structure of the input program—specifically, indentation of subprocedure calls under the printing of the procedure name which is doing the calling.

7. * PROGRAM CONTAINS UNBALANCED QUOTATION MARKS. * * CORRECT AND RESUBMIT FOR PROCESSING. * *

Meaning: This message is printed when the generator processes an input program that contains syntax errors of the type described in the error message itself. In addition to quotations marks, the errors may involve unbalanced apostrophes, parentheses, brackets, braces, and BEGIN-END instructions. The design of the generator is such that all characters between matching quotation marks, apostrophes, etc. are skipped over since procedure declarations and calls cannot occur in these areas. Also, the input code between matching BEGIN-END instructions is treated as separate and distinct blocks of code during
generator processing. Consequently, if any of these are unbalanced, the generator will erroneously process the remainder of the input program until an end-of-file is encountered. The generator will then recognize that an unbalanced situation exists. Processing ceases at this point, and the error message is printed. No other output is produced. The user must correct the problem and then resubmit his program for processing.
III. SYSTEM SPECIFICATIONS

Part III describes the general specifications for the procedure hierarchy generator. All figures in Part III are patterned after the technique used by the generator to display the structure of input programs. Specific line numbers of the source code are not shown, but these may be found in Appendix A and/or Appendix E.

System Overview

The procedure hierarchy generator is a stand-alone software package. The generator is operational on the KSU Interdata 8/32 computer which is housed in the minicomputer laboratory in Fairchild Hall. Coded in the sequential PASCAL language (the KSU implementation of sequential PASCAL), the generator consists of approximately 1200 lines of code, including imbedded comments. The code is well-structured, and extensive use of procedure definitions and calls keeps each code segment small and easily understandable. These properties enhance the ability to extend the generator with additional capabilities, as discussed later in this report.

Software Structure

The generator consists of a set of standard prefix declarations followed by the main program. The body of the
main program, termed "main routine", controls the sequence of processing. The main routine is preceded by the usual global constant, type, and variable declarations, and procedure routines (Figure III.1).

<table>
<thead>
<tr>
<th>STANDARD PREFIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>GENERATOR DECLARATIONS</td>
</tr>
<tr>
<td>CONSTANT</td>
</tr>
<tr>
<td>TYPE</td>
</tr>
<tr>
<td>VARIABLE</td>
</tr>
<tr>
<td>ROUTINE</td>
</tr>
<tr>
<td>MAIN ROUTINE</td>
</tr>
</tbody>
</table>

Figure III.1 Generator Program Structure

The generator routines can be logically divided into three functional groups. These groups and their constituent routines are listed below.

Initialization Routine.
INITIALIZE

Build Hash Table Routines.
BUILDTABLE
PARSE_FUNCTION
PARSE_PROCEDURE
PARSE_MAIN
ENTERTABLE
HASH_ENTER
APPEND_DETERMINE
HASH_APPEND
ALLOCATE_MORE_SUBPROC
APPENDTABLE

Print Hash Table Routines.
OUTPUT_CONTROL
TRAVERSE_TABLE
Hierarchically, then, the top level structure of the generator is as depicted in Figure III.2.

```
Main Routine.
  Initialization Routine.
  Build Hash Table Routines.
  Print Hash Table Routines.
 END.
```

Figure III.2 Generator Routines Structure

Two of the functional groups have extensive I/O processing. The Build Hash Table Routines handle input processing routines while the Print Hash Table Routines handle output processing routines as listed below in Figures III.3 and III.4. Both groups also handle error processing, if required.
Build Hash Table Routines.
GETWORD
GETCHAR
SKIP_COMMENT_APOSTROPHE
GETCHAR
UNBALANCED_ERROR
SKIP_PAREN_BRACKET
GETCHAR
UNBALANCED_ERROR
UNBALANCED_ERROR
END.

Figure III.3 Build Hash Table Routines Structure
(Input Routines)

Print Hash Table Routines.
WRIESTRING
HEADER
WRIESTRING
WRIENAME
WRIESTRING
EXCESS_PROCEDURES
WRIESTRING
EXCESS_NESTING
WRIESTRING
WRITEINT
WRIESTRING
END.

Figure III.4 Print Hash Table Routines Structure
(Output Routines)
Output Design

Appendix B contains the output syntax diagram for the generator. Table III.1 contains a representation of the two generator output formats. Refer to Appendix C for a detailed explanation of each column of output.

**TABLE III.1. Generator Output Formats**

<table>
<thead>
<tr>
<th>Seq</th>
<th>Ln No of the Main Program BEGIN or END Instruction</th>
<th>Column</th>
<th>Column</th>
<th>Column</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&quot;MAIN&quot; or &quot;END.&quot;</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Seq</td>
<td>Called Proc Declaration Line Number</td>
<td>a.</td>
<td>Called Proc Name</td>
<td>optional b.</td>
<td>optional c.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a. = Calling Statement Line Number (may be indented according to nested level of procedure)

b. = "[" Number of Consecutive Procedure Calls "]"

c. = "(" Repetitive Procedure Call Designation ")"

Data Structure

The data structure used in implementation of the generator is an array of records (specifically, a hash table). A node is created for each input program procedure during the parse of the program source code. Each node consists of: the node creation sequence number; a procedure name; an index to
continuation nodes, if required; a flag-field set when a
procedure is hierarchically expanded for the first time; a
program line number in which a procedure was declared; and a
count of the number of subprocedures called by a procedure,
if any. If subprocedures are called, then each node will
also have a variable number of sub-arrays. Each sub-array
consists of: the number of consecutive calls, if any; a
program line number where a subprocedure was called; and an
index which points to the location where the subprocedure
was entered into the hashtable as a procedure. The
generator output is formulated by traversal of the hash
table, using stored hash keys as pointers to the next
hashtable entry slot. Overflow of the procedure name length
(name > 20 characters) and/or the number of subprocedure
calls (calls > 20) is handled via use of the next available
blank storage location in the hash table. See Appendix D
for a layout of the implementation data structure.

An arbitrary 201 slots for the hash table has been
established to store information concerning procedure
declarations. This means that an input program can contain
200 procedure declarations and a main routine if the
following criteria are met:

- procedure names <= 20 characters in length.
- quantity of subprocedure calls within procedures
  and the main routine <= 20 calls.

For every instance where a procedure name contains 21 to 40
characters, the maximum number of procedure declarations is decreased by one because another slot has to be used to store the remainder of the procedure name. A 41 to 60 character name will decrease the slots available by two, and a 61 to 80 character name will decrease the slots available by 3.

Similarly, for every instance where a procedure contains 21 to 40 subprocedure calls, the maximum number of procedure declarations is decreased by one. Forty-one to 60 calls will decrease the slots by two, etc.

If an input program is so large that the generator cannot store all the procedure declarations and/or information on subprocedure calls, adjustment is quite simple. Depending on the reason for the excessiveness of the program, one of three actions can be taken:

- increase the number of characters allowed per slot per procedure name.
- increase the number of subprocedure calls allowed per slot per procedure.
- increase the number of hash table slots.

Any or all of these actions can be accomplished by making trivial changes in the global constant section of the generator. The generator would then, of course, have to be recompiled.
Routines

The following is a discussion of the processing logic comprising the functional groups of generator routines.

A. Initialization Routine. INITIALIZE is the first routine invoked by the main routine. INITIALIZE initializes each global variable to a pre-determined starting value.

B. Build Hash Table Routines.
   1. General. The Build Hash Table Routines read the input program code, character by character; analyze the code to detect procedure names (declarations) and procedure calls; and construct and input the hash table entries for later retrieval and output by the Print Hash Table Routines. Hierarchically, the structure of the Build Hash Table Routines is depicted in Figure III.5.

```
BUILDTABLE
  PARSE_FUNCTION
  UNBALANCED_ERROR
  PARSE_PROCEDURE; PARSE_MAIN
  ENTER_TABLE
  HASH_ENTER
  EXCESS_PROCEDURES
  UNBALANCED_ERROR
  APPEND_DETERMINE
  HASH_APPEND
  ALLOCATE_MORE_SUBPROC
  APPEND_TABLE
END.
```

Figure III.5 Structure of the Build Hash Table Routines
The Build Hash Table Routines are initiated by the main routine through a call to BUILDTABLE.

2. BUILDTABLE Routine. The BUILDTABLE routine controls parsing of the input program and controls construction of the hashtable. When function declarations are detected, BUILDTABLE transfers control to PARSE_FUNCTION; when procedure declarations are detected, it transfers control to PARSE_PROCEDURE; and when the main routine is detected, it transfers control to PARSE_MAIN. BUILDTABLE itself handles parsing of other portions of the input program, e.g., global type declarations and global variable declarations.

3. PARSE_PROCEDURE; PARSE_MAIN Routines. These routines function similarly. Their first action is to cause the procedure name to be entered into the hash table ("MAIN" in the case of PARSE_MAIN) via a call to ENTERTABLE. They then parse the local code, invoking APPEND_DETERMINE where a word encountered may be a procedure call. Termination of the local code is determined by matching "BEGIN" and "END" instructions. If matching "BEGIN" and "END" instructions are erroneously not present, the routines call an error routine (UNBALANCED_ERROR), and processing is terminated.

4. PARSE_FUNCTION Routine. PARSE_FUNCTION merely determines when the local code of a function has terminated. It then returns control to BUILDTABLE. Consequently,
procedure calls within functions are not detected. Termination of the local code is determined by matching "BEGIN" and "END" instructions. If matching "BEGIN" and "END" instructions are erroneously not present, the routine calls an error routine (UNBALANCED_ERROR), and processing is terminated.

5. ENTERTABLE Routine. ENTERTABLE obtains a hash key for the procedure name from the HASH_ENTER routine. With that hash key, it then enters the name into the hash table along with a nodenumber and the program line number of the procedure declaration.

6. HASH_ENTER Routine. HASH_ENTER computes a hash key based upon a maximum of the first 20 characters of the procedure name. If there are insufficient slots left in the hash table to enter another procedure name, then generator processing cannot continue. Control is then transferred to the EXCESS_PROCEDURES routine which is discussed under Error Processing later in this report. If the slot computed has been previously used, HASH_ENTER recomputes the key until an unused slot is found.

7. APPEND_DETERMINE Routine. APPEND_DETERMINE first determines if the input word being examined is a subprocedure call. It obtains a hash key for that word from the HASH_APPEND routine. It then matches that word with the word stored in the hash table that has the hash key just computed. If a match is found, the input word is a procedure call, and information must be appended to the
procedure currently being parsed. APPEND_DETERMINE then checks to see if an append slot is available. An initial allocation of 20 append slots exists. If, for example, a subprocedure call is the 21st for the procedure being parsed, then a call is made to the ALLOCATE_MORE_SUBPROC routine which gives the procedure being parsed another 20 append slots. This allocation process may be repeated as required. APPEND_DETERMINE then invokes APPENDTABLE.

8. APPENDTABLE Routine. APPENDTABLE determines whether the subprocedure call is an identical call to the last call made by the procedure being parsed or not. If it is an identical call, APPENDTABLE merely increments a consecutive call counter; if not, then appropriate information on the call is entered into the append slot, i.e., the program line number where the call occurred and the hash key of where that subprocedure itself is located in the hash table.

C. Print Hash Table Routines.

1. General. The Print Hash Table Routines are initiated by the main routine through a call to OUTPUT_CONTROL. The hash table entry for the main routine of the input program is located, and then the hash table is traversed and output via a recursive routine. Hierarchically, the structure of the Print Hash Table Routines is depicted in Figure III.6.
OUTPUT_CONTROL
TRVERSE_TABLE
END.

Figure III.6 Structure of the
Print Hash Table Routines

2. OUTPUT_CONTROL Routine. The OUTPUT_CONTROL
routine initially makes sure that the main routine of the
input program has been detected by BUILDTABLE. This is
necessary because of the possibility that the procedure
declaration just prior to the start of the main routine may
be of the form "PROCEDURE READ (C:CHAR);" with no local
code. If this occurs, the "BEGIN..END." of the main routine
will be initially interpreted as local code of the
procedure. Consequently, the hash table entries have to be
adjusted slightly. OUTPUT_CONTROL then outputs the first
line of the printed report which is the line number of the
main routine starting location. TRVERSE_TABLE is then
called. Upon return, OUTPUT_CONTROL outputs the last line
of the printed report which is the line number of the main
routine ending location and returns control to the generator
main routine.

3. TRVERSE_TABLE Routine. The TRVERSE_TABLE
routine traverses the hash table using stored hash keys to
navigate. Starting point for the traversal is the hash
table slot created for the main routine of the input
program. Data lines are created for the printed report as
the navigation progresses. TRAVERSE_TABLE is a recursive routine which greatly simplifies the navigation process. At this point in the processing, two errors can occur. A procedure may call excessive subprocedures (arbitrary limit established at 180 calls) or the input program may be structured so that procedure calls may be nested too deeply to allow complete depiction on a printout. In the first case, control is transferred to the EXCESS_PROCEDURES routine, and, in the second case, control is transferred to the EXCESS_NESTING routine. These error routines are discussed under Error Processing later in this report. In both cases, control is transferred back to TRAVERSE_TABLE, and processing continues.

Error Processing

A. Syntax/Semantic/Type Errors.

The generator is quite robust in that it will accept any input program and execute to a normal termination. However, it does not function similar to a compiler. If syntax errors exist in the input program, the generator will not detect these nor will it go to a resynchronizing point in the input code. For example, if a program with unbalanced quotation marks (delimiter for comments) is input to the generator, the only output created upon reaching the end of the input program will be an error message stating:
"INPUT PROGRAM CONTAINS AN UNBALANCED SET OF QUOTATION MARKS. CORRECT AND RESUBMIT YOUR PROGRAM FOR PROCESSING."

Similar error messages will be created for unbalanced sets of apostrophes, parentheses, brackets, braces, and BEGIN-END instructions. Handling of these conditions in this manner is necessary since the generator cannot determine where a comment, for example, ends and where the program instructions are supposed to begin again. A type error will not cause an error message to be output, but it may invalidate the output. Consider this partial program:

```pascal
VAR X : CHAR;
   .
   .
   .
PROCEDURE X (C : CHAR);
   .
   .
   .
   X := 'A';
   .
```

The user has declared a procedure named "X"; however, he has also declared a global variable "X" as type CHAR. Later, he assigns the variable "X" the value of "A". As the generator is parsing this program, it detects the "X" in the main program, finds this name in the hash table, and, consequently, makes the determination that this is a procedure call when, in fact, the user meant for it to be part of an assignment statement. The user is alerted to the possibility of this occurring through the header message which states:
WARNING: IF YOUR INPUT PROGRAM HAS COMPILATION
****** ERRORS IN PASS1 THROUGH PASS5, THE
****** FOLLOWING OUTPUT CANNOT BE CONSIDERED
****** RELIABLE.

Another example of a syntax error which will not cause an
error message to be output is the following:

PROCEDURE X (C : CHAR);
BEGIN
.
.
END;
BEGIN
XI(C);
END.

The user has declared a procedure named "X". He wants to
call that procedure in the main routine; however, he
mistakenly keys in "XI" instead of "X". The generator will
check to see if "XI" is a procedure name. Upon determining
that it is not, the generator will proceed on to the next
word in the input program, and the procedure call will not
be recognized.

B. Excessive Procedures.

The EXCESS_PROCEDURES routine provides error
messages to the user for both the occurrence of an excessive
number of procedure declarations and the occurrence of an
excessive number of subprocedure calls within a procedure or
within the main routine. If there are excessive procedure
declarations, then generator processing is halted, and a
message is output that the user should contact operations to
request that the number of hash table slots be increased and
that no output will be provided. If there are excessive subprocedure calls, then a message is output to the user advising him of this, and then processing resumes at the next logical point.

C. Excessive Nesting.

The EXCESS_NESTING routine provides an error message to the user during printing of the output whenever the indentations in the printed report, created by progressive nesting of procedure calls, would cause physical printing off the right side of the page. The generator can output nesting up to 19 levels deep. Nesting deeper than 19 levels will not be printed. Output will resume whenever nesting returns to a level no deeper than 19.

Future Extensions

During the design, coding, and testing of the generator, ideas continually surfaced concerning extensions to the original generator proposal. The generator, as implemented, does not provide any of the facilities discussed below, but each has sufficient merit to warrant inclusion in any future revisions of the generator.

A. Designation of procedure calls with conditional-type code and looping-type code.

The generator currently outputs procedure calls identically, regardless of whether the call is in
straight-line code, in conditional-type code (IF THEN ELSE constructs, CASE statements, etc.), or in looping-type code (FOR...DO, WHILE...DO, etc.). A distinction could be made between these types of procedure calls to more fully illustrate the input program structure.

B. User specification of maximum level of nesting that should be output.

The generator currently outputs all procedure calls (up to page print-out limitations). The capability could be provided to allow the user to specify that procedure calls only be output to a level of his choice. This would allow the user to control output quantity in cases where he is interested in only a portion of the input program structure.

C. Listing of procedures in the input program which are not called by any other procedure or the main routine.

The generator currently outputs only those procedures which are called. If a procedure is declared but not called, the user is not explicitly notified of this. The capability could be provided to list all such procedures. This would assist the user during program development activities.

D. Statistics/information summary.

The generator currently provides no statistics/information summary. The capability could be
provided to output a multitude of statistics/information, e.g., number of procedures in the input program, number of procedures called, number of procedures not called, maximum nesting level, most active procedures, etc.

E. Inclusion of the analysis of functions.

The generator currently disregards the local code of input program functions. This code may contain calls to procedures. The capability could be provided, perhaps as an option, to analyze this code and to produce output for functions that is similar to output produced for procedures and the main routine.
IV. BIBLIOGRAPHY


PROCEDURE HIERARCHY GENERATOR OUTPUT LISTING

SEQUENTIAL PASCAL PROCEDURE HIERARCHY GENERATOR

WARNING: IF YOUR INPUT PROGRAM HAS COMPILATION
******* ERRORS IN PASS1 THROUGH PASS5, THE
******* FOLLOWING OUTPUT CANNOT BE CONSIDERED
******* RELIABLE.

1  1178 MAIN
2  1112 1179 INITIALIZE
3   198  1129 WRITESTRING
4    82    212 WRITE
5    82    1130 WRITE
6   906  1180 BUILDTABLE
7   461    922 GETWORD
     355         478 GETCHAR
     81         371 READ
   10  404    479 SKIP_COMMENT_APOSTROPHE
   11  355    414 GETCHAR (8)
   12  375    419 UNBALANCED_ERROR
   13  198    388 WRITESTRING [7] (3)
   14    82    397 WRITE
   15  198    398 WRITESTRING (3)
   16    82    399 WRITE [2]
   17  355    423 GETCHAR (8)
   18  355    482 GETCHAR (8)
   19  404    483 SKIP_COMMENT_APOSTROPHE (10)
   20  427    488 SKIP_PAREN_BRACKET_BRACE [3]
   21  355    438 GETCHAR (8)
   22  375    443 UNBALANCED_ERROR (12)
   23  355    448 GETCHAR [2] (8)
   24  355    500 GETCHAR [2] (8)
   25  868    924 PARSE_FUNCTION
   26  461    879 GETWORD [2] (7)
   27  375    901 UNBALANCED_ERROR (12)
   28  765    930 PARSE_PROCEDURE
   29  461    777 GETWORD (7)
   30  725    788 ENTERTABLE
   31  668    740 HASH_ENTER
   32  251    719 EXCESS_PROCEDURES
   33  198  261 WRITESTRING[3](3)
   34    82    264 WRITE
   35  198  266 WRITESTRING[3](3)
   36    82    269 WRITE
   37  198    270 WRITESTRING (3)
   38    82    271 WRITE
   39  198    272 WRITESTRING (3)
   40    82    273 WRITE
   41  198    275 WRITESTRING[2](3)

APPENDIX A
<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>82</td>
<td>WRITE [ 2]</td>
</tr>
<tr>
<td>43</td>
<td>461</td>
<td>GETWORD [ 3] ( 7)</td>
</tr>
<tr>
<td>44</td>
<td>617</td>
<td>APPEND_DETERMINE</td>
</tr>
<tr>
<td>45</td>
<td>581</td>
<td>HASH_APPEND</td>
</tr>
<tr>
<td>46</td>
<td>557</td>
<td>ALLOCATE_MORE_SUBPROC</td>
</tr>
<tr>
<td>47</td>
<td>528</td>
<td>APPENDTABLE</td>
</tr>
<tr>
<td>48</td>
<td>557</td>
<td>ALLOCATE_MORE_SUBPROC</td>
</tr>
<tr>
<td>49</td>
<td>528</td>
<td>APPENDTABLE</td>
</tr>
<tr>
<td>50</td>
<td>375</td>
<td>UNBALANCED_ERROR ( 12)</td>
</tr>
<tr>
<td>51</td>
<td>831</td>
<td>PARSE_MAIN</td>
</tr>
<tr>
<td>52</td>
<td>725</td>
<td>ENTERTABLE ( 30)</td>
</tr>
<tr>
<td>53</td>
<td>461</td>
<td>GETWORD ( 7)</td>
</tr>
<tr>
<td>54</td>
<td>617</td>
<td>APPEND_DETERMINE ( 44)</td>
</tr>
<tr>
<td>55</td>
<td>375</td>
<td>UNBALANCED_ERROR ( 12)</td>
</tr>
<tr>
<td>56</td>
<td>1063</td>
<td>OUTPUT_CONTROL</td>
</tr>
<tr>
<td>57</td>
<td>302</td>
<td>HEADER</td>
</tr>
<tr>
<td>58</td>
<td>82</td>
<td>WRITE</td>
</tr>
<tr>
<td>59</td>
<td>198</td>
<td>WRITESTRING ( 3)</td>
</tr>
<tr>
<td>60</td>
<td>82</td>
<td>WRITE</td>
</tr>
<tr>
<td>61</td>
<td>198</td>
<td>WRITESTRING ( 3)</td>
</tr>
<tr>
<td>62</td>
<td>82</td>
<td>WRITE</td>
</tr>
<tr>
<td>63</td>
<td>198</td>
<td>WRITESTRING ( 3)</td>
</tr>
<tr>
<td>64</td>
<td>82</td>
<td>WRITE</td>
</tr>
<tr>
<td>65</td>
<td>198</td>
<td>WRITESTRING ( 3)</td>
</tr>
<tr>
<td>66</td>
<td>82</td>
<td>WRITE [ 2]</td>
</tr>
<tr>
<td>67</td>
<td>668</td>
<td>HASH_ENTER ( 31)</td>
</tr>
<tr>
<td>68</td>
<td>218</td>
<td>WRITEINT [ 2]</td>
</tr>
<tr>
<td>69</td>
<td>82</td>
<td>WRITE [ 5]</td>
</tr>
<tr>
<td>70</td>
<td>198</td>
<td>WRITESTRING ( 3)</td>
</tr>
<tr>
<td>71</td>
<td>82</td>
<td>WRITE</td>
</tr>
<tr>
<td>72</td>
<td>947</td>
<td>TRAVERSE_TABLE</td>
</tr>
<tr>
<td>73</td>
<td>251</td>
<td>EXCESS_PROCEDURES ( 32)</td>
</tr>
<tr>
<td>74</td>
<td>218</td>
<td>WRITEINT [ 2] ( 68)</td>
</tr>
<tr>
<td>75</td>
<td>198</td>
<td>WRITESTRING ( 3)</td>
</tr>
<tr>
<td>76</td>
<td>218</td>
<td>WRITEINT ( 68)</td>
</tr>
<tr>
<td>77</td>
<td>173</td>
<td>WRITENAME [ 2]</td>
</tr>
<tr>
<td>78</td>
<td>82</td>
<td>WRITE</td>
</tr>
<tr>
<td>79</td>
<td>198</td>
<td>WRITESTRING ( 3)</td>
</tr>
<tr>
<td>80</td>
<td>218</td>
<td>WRITEINT ( 68)</td>
</tr>
<tr>
<td>81</td>
<td>198</td>
<td>WRITESTRING [ 2] ( 3)</td>
</tr>
<tr>
<td>82</td>
<td>218</td>
<td>WRITEINT ( 68)</td>
</tr>
<tr>
<td>83</td>
<td>198</td>
<td>WRITESTRING ( 3)</td>
</tr>
<tr>
<td>84</td>
<td>82</td>
<td>WRITE</td>
</tr>
<tr>
<td>85</td>
<td>198</td>
<td>WRITESTRING ( 3)</td>
</tr>
<tr>
<td>86</td>
<td>82</td>
<td>WRITE</td>
</tr>
<tr>
<td>87</td>
<td>198</td>
<td>WRITESTRING ( 3)</td>
</tr>
<tr>
<td>88</td>
<td>82</td>
<td>WRITE</td>
</tr>
<tr>
<td>89</td>
<td>947</td>
<td>TRAVERSE_TABLE ( 72)</td>
</tr>
<tr>
<td>90</td>
<td>283</td>
<td>EXCESS_NESTING</td>
</tr>
<tr>
<td>91</td>
<td>198</td>
<td>WRITESTRING ( 3)</td>
</tr>
<tr>
<td>92</td>
<td>82</td>
<td>WRITE</td>
</tr>
<tr>
<td>93</td>
<td>198</td>
<td>WRITESTRING ( 3)</td>
</tr>
<tr>
<td>94</td>
<td>82</td>
<td>WRITE</td>
</tr>
<tr>
<td>95</td>
<td>198</td>
<td>WRITESTRING ( 3)</td>
</tr>
</tbody>
</table>
296  WRITE
297  WRITESTRING ( 3)
298  WRITE
1104  WRITEINT [ 2] ( 68)
1107  WRITESTRING ( 3)
1108  WRITE
1186  TRAILER [ 2]
331  WRITESTRING ( 3)
332  WRITE
333  WRITESTRING [ 2] ( 3)
336  WRITE [ 2]
1189  WRITE
1191  END.
OUTPUT SYNTAX DIAGRAM

--- > Ln No of --- > "END." --- >
Last END
Inst in
Main Routine

--- > Seq No --- > Called Proc Dcl Ln No

--- > Calling --- > Called
Statement Proc
Ln No Name

<-- "MAIN" <-- Ln No of <--
1st BEGIN
Inst in
Main Routine

--- > "[" --> No Of --> "]" --> "(" --> Repetitive --> ")" ->
Consec Proc Calls
Proc Call Designation

APPENDIX B
OUTPUT SPECIFICATIONS

Column 1 - Sequence numbers.

1. Purpose - ordering of the procedure calls.
   - output table entry point for subsequent referrals to repetitive procedure calling sequences.

2. Value - 1..9999.

3. Format - maximum of four numeric characters, right-justified.

Column 2 - Called Procedure Declaration Line Numbers.

1. Purpose - to denote the line number to which execution flow has been transferred by a procedure call.

2. Value - program line number of the declaration statement of the called procedure.

3. Format - maximum of five numeric characters, right-justified.

Column 3 - Calling Statement Line Numbers.

1. Purpose - to denote the line number of the statement from which execution flow has been transferred.

2. Value - program line number of the statement which is invoking a procedure call.

3. Format - maximum of five numeric characters, left-justified.

Column 4 - Called Procedure Names.

1. Purpose - to denote the name of the procedure to which execution flow has been transferred.

2. Value - name of the procedure being invoked.

3. Format - standard PASCAL procedure naming syntax, left-justified.

APPENDIX C
Column 5 - Consecutive Procedure Calls.

1. **Purpose** - to decrease output quantity and to increase output readability and efficiency. If a procedure is called twice consecutively, for example, there is no viable reason to hierarchically expand each call and output the structure twice.

2. **Value** - number of consecutive procedure calls made to the called procedure currently being analyzed.

3. **Format** - maximum of two numeric characters, enclosed in brackets. Example = [23].

Column 6 - Repetitive Procedure Calls.

1. **Purpose** - to decrease output quantity and increase output readability and efficiency. Once the hierarchy of calls within procedures has been initially determined and output, there is no viable reason to re-determine the structure for each subsequent call and to output the structure more than once. All that is required is a reference (a sequence number) to the pertinent structure in the display printed previously.

2. **Value** - a sequence number previously generated during a hierarchical expansion of a procedure call.

3. **Format** - maximum of three numeric characters, enclosed in parentheses. Example = (20); by referring to the sequence number in parentheses, the user can determine the hierarchy of the procedure currently being examined.
IMPLEMENTATION DATA STRUCTURE

<table>
<thead>
<tr>
<th>Node No</th>
<th>Proc Name</th>
<th>Link Key</th>
<th>Repetitive Ind</th>
<th>Proc Dcl Ln No</th>
<th>No of Calls</th>
<th>No of Consec Calls</th>
<th>Ln No of Subproc Call</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>9999</td>
<td>20(X)</td>
<td>999</td>
<td>999</td>
<td>9999</td>
<td>999</td>
<td>999</td>
<td>9999</td>
<td>999</td>
</tr>
</tbody>
</table>

Translation Key -

Node No : Node creation sequence number.
Proc Name : Procedure name.
Link Key : Index to continuation nodes.
Repetitive Ind : Repetitive procedure expansion indicator.
Proc Dcl Ln No : Procedure declaration line number.
Subproc Calls : Number of subprocedure calls.
No of Consec Calls : Number of consecutive subprocedure calls.
Ln No of Subproc Call : Program line number where subprocedure was called.
Subproc Key : Index to subprocedure declaration node.
9 : Numeric character.
X : Alphanumeric character.
"PER BRINCH HANSEN
INFORMATION SCIENCE
CALIFORNIA INSTITUTE OF TECHNOLOGY
UTILITY PROGRAMS FOR
THE SOLO SYSTEM
18 MAY 1975"

###

###

```plaintext
CONST NL = '(*101)';  FF = '(*121)';  CR = '(*131)';
               EN = '(*125)';

CONST PAGELENGTH = 512;
TYPE PAGE = ARRAY (.1..PAGELENGTH.) OF CHAR;

CONST LINELENGTH = 132;
TYPE LINE = ARRAY (.1..LINELENGTH.) OF CHAR;

CONST IDLENGTH = 12;
TYPE IDENTIFIER = ARRAY (.1..IDLENGTH.) OF CHAR;

TYPE FILE = 1..2;

TYPE FILEKIND = (EMPTY, SCRATCH, ASCII, SEQCODE, CONCODE);

TYPE FILEATTR = RECORD
  KIND: FILEKIND;
  ADDR: INTEGER;
  PROTECTED: BOOLEAN;
  NOTUSED: ARRAY (.1..5.) OF INTEGER
END;

TYPE IODEVICE =
  (TYPEDEVICE, DISKDEVICE, TAPEDEVICE, PRINTDEVICE, CARUDEVICE);

TYPE IOPERATION = (INPUT, OUTPUT, MOVE, CONTROL);

TYPE IODARG = (WRITEEOF, REWIND, USPACE, BACKSPACE);

TYPE IORESULT =
  (COMPLETE, INTERVENTION, TRANSMISSION, FAILURE,
   ENDFILE, ENDMEDIUM, STARTMEDIUM);

TYPE IOPARAM = RECORD
  OPERATION: IOPERATION;
  STATUS: IORESULT;
END;
```

E-2
ARG: IOARG
END:

TYPE TASKKIND = (INPUTTASK, JOBTASK, OUTPUTTASK);

TYPE ARGTAG = (NILTYPE, BOOLEAN, INTTYPE, IDTYPE, PTRTYPE);

TYPE POINTER = BOOLEAN;

TYPE ARGTYPE = RECORD
  CASE TAG: ARGTAG OF
  NILTYPE: BOOLEAN: (T: BOOLEAN);
  INTTYPE: INT: INTEGER;
  IDTYPE: ID: IDENTIFIER;
  PTRTYPE: PTR: POINTER;
END;

CONST MAXARG = 10;

TYPE ARGLIST = ARRAY [1..MAXARG] OF ARGTYPE;

TYPE ARGSEQ = (INPUT, OUTPUT);

TYPE PROGRESS = (TERMINATED, OVERFLOW, POINTERERROR, RANGEERROR, VARIANTERROR, 
  HEAPLIMIT, STACKLIMIT, CODELIMIT, TIMELIMIT, CALLERROR);

PROCEDURE READ(var C: CHAR);
PROCEDURE WRITE(var C: CHAR);
PROCEDURE OPEN(var F: FILE; ID: IDENTIFIER; VAR FOUND: BOOLEAN);
PROCEDURE CLOSE(var F: FILE);
PROCEDURE SET(var F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);
PROCEDURE PUT(var F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);
FUNCTION LENGTH(var F: FILE): INTEGER;
PROCEDURE MARK(var TOP: INTEGER);
PROCEDURE RELEASE(var TOP: INTEGER);
PROCEDURE IDENTIFY(HEADER: LINE);
PROCEDURE ACCEPT(var C: CHAR);
PROCEDURE DISPLAY(var C: CHAR);

PROGRAM GEN;

"*****************************************************************************
  * GENERATOR CONSTANT, TYPE, AND VARIABLE DECLARATIONS
  *****************************************************************************

CONST
  MAXWORDLENGTH = 20;
  MAXSTRINGLENGTH = 60;

C-3
MAXUNIQUEPROCEDURES = 201;
HASHMAXI = 201;
MAXPROCEDURECALLS = 201;
MAXCALLS1 = 201;
SPAN = 271;
MAXWORDBYTE = 4;
END_STRING = '(':);'
QUOTEMARK = '(:);'
APOSTROPHE = '(:);';
LEFT_PAREN = '{(:);};'
RIGHT_PAREN = '(:);';
LEFT_BRACKET = '[:(:);];'
RIGHT_BRACKET = '[:(:);];'
LEFT_BRACE = '[:(:);];'
RIGHT_BRACE = '[:(:);];'
DESIN = 'BEGIN';
END = 'END';
FUNCTION = 'FUNCTION';
PROCEDURE = 'PROCEDURE';
MAIN = 'MAIN';
VAR = 'VAR';
FORWARD = 'FORWARD';
TYPE = 'TYPE';
CONST = 'CONST';
CASE = 'CASE';
EXTERN = 'EXTERN';

TYPE
LINES = ARRAY[1..MAXSTRINGLENGTH] OF CHAR;
WORDINDEX = 1..MAXWORDBYTE;
WORD = ARRAY[WORDINDEX] OF CHAR;
WORDTYPE = ARRAY[1..MAXWORDBYTE] OF WORD;
HASHKEY = 0..MAXUNIQUEPROCEDURES;
SUBPROC = ARRAY[1..MAXPROCEDURECALLS] OF
RECORD
SUBPROC_CONSECALLS : SHORTINTEGER;
SUBPROC_LINENO : SHORTINTEGER;
SUBPROC_HASHKEY : SHORTINTEGER;
END;
HASH_TABLE = ARRAY[HASHKEY] OF
RECORD
NOGENUMBER : SHORTINTEGER;
PROGRAMNAME : WORD;
LINKKEY : SHORTINTEGER;
REPETITIVE : SHORTINTEGER;
PROC_LINENO : SHORTINTEGER;
NO_SUBPROC_CALLS : SHORTINTEGER;
NESTED : SUBPROC;
END;

VAR
NULL, ENTER_COUNT, CURRENTLINE : SHORTINTEGER;
EXCESS_CALL_FLAG, MAIN_FLAG, JUSTIFY, CALL_FLAG,
PROCEDURE_FLAG, HALT_FLAG : BOOLEAN;
HOLD_LINK_KEY, HOLD_HASHKEY : HASHKEY;
LETTERS, NUMBERS, LETTERS_NUMBERS : SET OF CHAR;
CURRENTCHAR : CHAR;
HASHTABLE : HASH_TABLE;
CURRENTWORD, CONSEC_COMPARE : WORDTYPE;

*******************************************************************************
+ GENERATOR PROCEDURE DECLARATIONS +
*******************************************************************************

*******************************************************************************
PROCEDURE WRITENAME (C : WORD; SPACER : SHORTINTEGER) : 
"GLOBAL VARIABLES REFERENCED : NONE.
"CALLING MODULES : TRAVERSE_TABLE

VAR
I : WORDINDEX;
PR_COUNT_CHAR : SHORTINTEGER;
BEGIN
I := 11
PR_COUNT_CHAR := SPACER * 6 + 18;
WHILE I <= MAXWORDLENGTH DO
BEGIN
IF C[I] <> ' ' THEN
BEGIN
PR_COUNT_CHAR := PR_COUNT_CHAR + 1;
IF PR_COUNT_CHAR < 133 THEN WRITE (C[I])
END;
I := I + 1
END;
END;

*******************************************************************************
PROCEDURE WRITESSTRING (TEXT : LINES) : 
"GLOBAL VARIABLES REFERENCED : NONE.
"CALLING MODULES : EXCESS_NESTING / OUTPUT_CONTROL/ 
"EXCESS_PROCEDURES / HEADER / 
"TRAVERSE_TABLE / THAILER / 
"UNBALANCED_ERROR / INITIALIZE / 

VAR
I : SHORTINTEGER;
BEGIN
I := 11
WHILE TEXT[I] <> END_STRING DO
BEGIN
WRITE (TEXT[I]):
I := I + 1
END
END;
E-5
PROCEDURE WRITEINT (N, DIGITS : INTEGER);

"PURPOSE : WRITES INTEGERS TO LOGICAL UNIT 2"

"GLOBAL VARIABLES REFERENCED : JUSTIFY"

"CALLING MODULES : TRAVERSE_TABLE/ OUTPUT_CONTROL/

VAR

REM, DIG, I : 1..6;
NUM : ARRAY[1..6] OF CHAR;
BEGIN
REM := N;
DIG := 1;
REPEAT
NUM[ DIG ] := CHAR(ABS(REM) MOD 10 + ORD( '0' ));
REM := REM DIV 10;
DIG := SUCCEED
UNTIL REM = 0;
IF NOT JUSTIFY
THEN
FOR I := DIGS DOWTO 1 DO
IF I >= DIG
THEN WRITE( ' ' )
ELSE WRITE( NUM[I] )
ELSE
BEGIN
WRITE( ' ' );
FOR I := DIG - 1 DOWTO 1 DO
WRITE( NUM[I] );
FOR I := DIG TO DIGS DO
WRITE( ' ' )
END
END;

PROCEDURE EXCESS_PROCEDURES;
"PURPOSE : OUTPUT USER MESSAGE IF THE PROGRAM BEING PARSED"
"HAS TOO MANY PROCEDURES FOR THE HASTABLE TO STORE"
"OR THE PROCEDURE BEING PARSED HAS TOO MANY"
"SUBPROCEDURES FOR THE HASTABLE TO STORE"
"GLOBAL VARIABLES REFERENCED : HALT_FLAG"
"CALLING MODULES : TRAVERSE_TABLE/ HASHENTER/

BEGIN
IF HAIL_FLAG
THEN WRITESTRING ( ' THE PROGRAM(10)' );
ELSE WRITESTRING ( ' THE PROGRAM(10)' );
WRITESTRING ( ' BEING ANALYZED HAS TOO MANY(10)' );
WRITE( INL );
IF HAIL_FLAG
THEN WRITESTRING ( ' PROCEDURES(10)' );
ELSE WRITESTRING ( ' SUBPROCEDURE CALLS(10)' );
WRITESTRING ( ' FOR THE GENERATOR, AS(10)' );
WRITE( INL )
E-6
WRITESTRING (* CURRENTLY SET UP, CONTACT OPERATIONS TO:(0:1)*):
WRITE (NL):
WRITESTRING (* REQUEST THAT THIS ARBITRARY LIMIT BE RAISED:(0:1)*):
WRITE (NL):
IF HALT_FLAG
  THEN WRITESTRING (* NO GENERATOR OUTPUT CAN BE PROVIDED:(0:1)*):
ELSE WRITESTRING (* THIS PROCEDURE WILL NOT BE ANALYZED FURTHER:(0:1)*):
WRITE (NL):
IF HALT_FLAG
  THEN WRITE (EM)
END:

*********************************************************************************************
PROCEDURE EXCESS_NESTING:
"PURPOSE : OUTPUT USER MESSAGE THAT THE PROCEDURE BEING PARSED:
"HAS A NESTING LEVEL SO DEEP THAT IT CANNOT BE
"DISPLAYED ON A PRINTED PAGE
"GLOBAL VARIABLES REFERENCED : NONE.
"CALLING MODULES : TRAVERSE_TABLE/
BEGIN
WRITESTRING(* THE PROCEDURE CURRENTLY BEING EXPANDED HAS A:(0:1)*):
WRITE(NL):
WRITESTRING(* NESTING LEVEL DEEPER THAN 19. THIS EXCEEDS:(0:1)*):
WRITE(NL):
WRITESTRING(* THE PHYSICAL PAPER OUTPUT LIMITATIONS. THE:(0:1)*):
WRITE(NL):
WRITESTRING(* PROCEDURE CALLS WILL NOT BE EXPANDED FURTHER:(0:1)*):
WRITE(NL)
END:

*********************************************************************************************
PROCEDURE HEADER:
"PURPOSE : WRITES OUT A HEADER MESSAGE PRIOR TO OUTPUTTING DATA
"GLOBAL VARIABLES REFERENCED : NONE.
"CALLING MODULES : OUTPUT-CONTROL/
BEGIN
WRITE (NL):
WRITESTRING (*) WARNING : IF YOUR INPUT PROGRAM HAS COMPILATION:(0:1)*):
WRITE (NL):
WRITESTRING (*) ERROR IN PASS 1 THROUGH PASS 2: THE:(0:1)*):
WRITE (NL):
WRITESTRING (*) FOLLOWING OUTPUT CANNOT BE CONSIDERED:(0:1)*):
WRITE (NL):
WRITESTRING (*) RELIABLE.(0:1)*):
WRITE (NL)
END:

*********************************************************************************************
PROCEDURE TRAILER:
"PURPOSE : WRITES OUT A TRAILER MESSAGE SUBSEQUENT TO
"OUTPUTTING DATA, IF REQUIRED
C=7
"GLOBAL VARIABLES REFERENCED : PROCEDUREFLAG
"CALLING MODULES : MAIN

BEGIN
  IF PROCEDUREFLAG
    THEN
      BEGIN
        WRITESTRING ('***** PROGRAM CONTAINS PROCEDURES: HOWEVER, *****(0):')
        WRITE (NL)
        WRITESTRING (' MAIN DOES NOT INVOKES ANY OF THEM', (0):')
      END
    ELSE WRITESTRING ('***** PROGRAM CONTAINS NO PROCEDURES. *****(0):')
    WRITE (NL)
    WRITE (EM)
  END:

FUNCTION EOF : BOOLEAN:

BEGIN
  EOF := CURRENTCHAR = EM
END:

FUNCTION COLN : BOOLEAN:

BEGIN
  COLN := CURRENTCHAR = NL
END:

PROCEDURE GETCHAR:
  "PURPOSE : READ A SINGLE CHARACTER FROM THE INPUT FILE"
  "GLOBAL VARIABLES REFERENCED : CURRENTCHAR, CURRENTLINE"
  "CALLING MODULES : SKIP_COMMENT_APOSTROPH ( / GETWORD /"
  " / SKIP_PARENTHESES /"

BEGIN
  IF EOF
    THEN CURRENTCHAR := EM
  ELSE
    IF COLN
      THEN BEGIN
        CURRENTCHAR := ' '
        CURRENTLINE := CURRENTLINE + 1
      END
    ELSE READ (CURRENTCHAIN)
END:

PROCEDURE UNBALANCED_ERROR (CHAR1 : CHAR):
  "PURPOSE : PRINTS OUT AN ERROR MESSAGE TO USERS IF THE INPUT"
  " PROGRAM CONTAINS ANY UNBALANCED QUOTATION MARKS."

E-8
BEGIN
   WRITestring ("INPUT PROGRAM CONTAINS AN UNBALANCED SET OF ( inflammation);"
   CASE CH1 OF
      QUOTEMARK: WRITestring (" QUOTATION MARKS. ( inflammation;"
      APOSTROPHE : WRITestring (" APOSTROPHES. ( inflammation;"
      LEFT_PAREN : WRITestring (" PARENTHESES. ( inflammation;"
      LEFT_BRACKET : WRITestring (" BRACKETS. ( inflammation;"
      LEFT_BRACE : WRITestring (" BRACES. ( inflammation;"
      END_STRING: WRITestring (" BEGIN-ENDS. ( inflammation;"
   END;
   WRITestring (" CORRECT AND RESUBMIT YOUR PROGRAM FOR PROCESSING. ( inflammation;"
   WRITE (NL):
   WRITE (EM)
   END;

PROCEDURE SKIP_COMMENT_APOSTROPE (CHAR1 : CHAR):"*
   "PURPOSE : SKIPS OVER CHARACTERS ENCLOSED IN QUOTEMARKS AND APOSTROPHES"
   "GLOBAL VARIABLES REFERENCED : CURRENTCHAR"
   "CALLING MODULES : GETWORD"
BEGIN
   IF CURRENTCHAR <> CHAR1
      THEN
         REPEAT
            GETCHAR:
            IF EOF
               THEN
                  BEGIN
                      HALT_FLAG := TRUE;
                      UNBALANCED_ERROR (CHAR1)
                  END
            UNTIL (CURRENTCHAR = CHAR1) OR (EOF);
   IF NOT HALT_FLAG
   THEN GETCHAR
   END:
PROCEDURE SKIP_PAREN_BRACKET_BRACE (CHAR1, CHAR2 : CHAR):
   "PURPOSE : SKIPS OVER CHARACTERS ENCLOSED IN PARENTHESES, BRACKETS, AND BRACES"
   "GLOBAL VARIABLES REFERENCED : CURRENTCHAR"
   "CALLING MODULES : GETWORD"
VAR
I : SHORTINTEGER;
BEGIN
I := 1;
REPEAT
GETCHAR;
IF EOF THEN
BEGIN
HALT_FLAG := TRUE;
UNBALANCED_ERROR (CHAR1)
END:
IF CURRENTCHAR = APOSTROPHE THEN
BEGIN
GETCHAR;
WHILE (CURRENTCHAR <> APOSTROPHE) AND NOT (EOF) DO
GETCHAR
END:
IF CURRENTCHAR = CHAR1 THEN I := I + 1
ELSE IF CURRENTCHAR = CHAR2 THEN I := I - 1
UNTIL (I = 0) OR (HALT_FLAG)
END:

********************************************************************************

PROCEDURE GETWORD:

"PURPOSE : READS A SINGLE WORD FROM THE INPUT FILE"
"GLOBAL VARIABLES REFERENCED : CURRENTCHAR, LETTERS / NUMBERS / CURRENTWORD/
"CALLING MODULOS : PARSEPROCEDURE, PARSEMAIN/

VAR
INDEX, BLANKINDEX : 0..MAXWORDLENGTH;
I, X : SHORTINTEGER;
CHAR1, CHAR2 : CHAR;
BEGIN
LETTERS_NUMBERS := LETTERS - NUMBERS;
WHILE NOT (EOF OR (CURRENTCHAR IN LETTERS_NUMBERS)) DO
CASE CURRENTCHAR OF
QUOTEMARK : BEGIN
GETCHAR;
SKIP_COMMENT_APOSTROPHE (QUOTEMARK)
END:
APOSTROPHE : BEGIN
GETCHAR;
SKIP_COMMENT_APOSTROPHE (APOSTROPHE)
END:
LEFT_PAREN : BEGIN
E-10
CHAR1 := LEFT_PAREN:
CHAR2 := RIGHT_PAREN:
SKIP_PAREN_BRACKET_BRACE (CHAR1, CHAR2): END:

LEFT_BRACKET : BEGIN
CHAR1 := LEFT_BRACKET:
CHAR2 := RIGHT_BRACKET:
SKIP_PAREN_BRACKET_BRACE (CHAR1, CHAR2):
END:

LEFT_BRACE : BEGIN
CHAR1 := LEFT_BRACE:
CHAR2 := RIGHT_BRACE:
SKIP_PAREN_BRACKET_BRACE (CHAR1, CHAR2):
END:
ELSE
BEGIN
I := 0:
WHILE CURRENTCHAR IN LETTERS_NUMBERS DO
BEGIN
LETTERS_NUMBERS := LETTERS + NUMBERS:
INDEX := 0:
I := I + 1:
WHILE (CURRENTCHAR IN LETTERS_NUMBERS) AND
(INDEX <> MAXWORDLENGTH) DO
BEGIN
INDEX := INDEX + 1:
CURRENTWORD1.INDEX1 := CURRENTCHAR:
GETCHAR:
END:
IF INDEX < MAXWORDLENGTH
THEN
FOR BLANKINDEX := INDEX + 1 TO MAXWORDLENGTH DO
CURRENTWORD1.BLANKINDEX1 := ' ':
END:
IF I < MAXWORDTYPE
THEN
FOR X := I + 1 TO MAXWORDTYPE DO
FOR BLANKINDEX := 1 TO MAXWORDLENGTH DO
CURRENTWORD1.BLANKINDEX1 := ' ':
END:
BEGIN
IF CURRENTWORD11 <> CONSEC_COMPARE11
THEN
E-11
BEGIN
    CALLCOUNT := CALLCOUNT + 1;
    WITH HASHTABLE(EXCESS_KEY1,NESTEDCALLCOUNT) DO BEGIN
        SUBPROC_LINENO := CURRENTLINE;
        SUBPROC_HASHKEY := APPEND_KEY
        END;
        WITH HASHTABLE(HOLD_HASHKEY) DO
            NO_SUBPROC_CALLS := NO_SUBPROC_CALLS + 1;
            CONSEC_COMPARE := CURRENTWORD
        END;
        ELSE
            WITH HASHTABLE(EXCESS_KEY1,NESTEDCALLCOUNT) DO
                SUBPROC_CONSECALLS := SUBPROC_CONSECALLS + 1
            END;

END;

********************************************************************************
PROCEDURE ALLOCATE_MORE_SUBPROC (VAR CALLCOUNT : SHORTINT; KEY : HASHTABLE;)
VAR EXCESS_KEY : HASHTABLE;
"PURPOSE : OBTAIN AN ADDITIONAL HASHTABLE ENTRY POINT FOR"
"MORE SUBPROCEDURE CALL INFORMATION"
"GLOBAL VARIABLES REFERENCED : HOLD_LINK_KEY/ NULL/"
"HASHTABLE/"
"CALLING MODULES : APPEND_DETERMINE/
BEGIN
    CALLCOUNT := 0;
    IF HOLD_LINK_KEY = MAXUNIQUEPROCEDURES
        THEN EXCESS_KEY := 0
    ELSE EXCESS_KEY := HOLD_LINK_KEY + 1;
    WHILE (HASHTABLE(EXCESS_KEY1),NODENUMBER <> NULL DO
        IF EXCESS_KEY = MAXUNIQUEPROCEDURES
            THEN EXCESS_KEY := 0
        ELSE EXCESS_KEY := EXCESS_KEY + 1;
        WITH HASHTABLE(EXCESS_KEY1) DO
            NODENUMBER := NODENUMBER + 1;
            HASHTABLE(HOLD_LINK_KEY1,LINK_KEY := EXCESS_KEY;
        HOLD_LINK_KEY := EXCESS_KEY
    END;

********************************************************************************
PROCEDURE HASH_APPEND (VAR KEY : HASHKEY; VAR SEARCH : BOOLEAN;)
VAR FOUND : BOOLEAN;
"PURPOSE : DETERMINE THE HASHKEY FOR THE SUBPROCEDURE CALLED BY THE PROCEDURE BEING PARSED"
"GLOBAL VARIABLES REFERENCED : CURRENTWORD/ HASHTABLE/"
"CALLING MODULES : APPEND_DETERMINE/"
VAR
    CHAR_INDEX : WORDINDEX;
BEGIN
    KEY := 1;
    FOR CHAR_INDEX := 1 TO MAXWORDLENGTH DO
        IF CURRENTWORD1(CHAR_INDEX) <> ''
    C-12
THEN KEY := KEY * (ORD(CURRENTWORD[1], CHAK_INDEX) MOD SPAN)
IF HASHTABLE[KEY,NODENUMBER] = NULL
  BEGIN
    SEARCH := FALSE;
    FOUND := FALSE
  END
ELSE
  IF HASHTABLE[KEY, PROCNAME] = CURRENTWORD[1]
    THEN
      BEGIN
        FOUND := TRUE;
        SEARCH := FALSE
      END
    ELSE
      BEGIN
        SEARCH := TRUE;
        FOUND := FALSE
      END
   END

******************************************************************************
PROCEDURE APPEND_DETERMINE (VAR CALLCOUNT : SHORTINTEGER);
  "PURPOSE : DETERMINE IF THE CURRENTWORD IS A PROCEDURE"
  " CALL WHICH SHOULD BE APPENDED TO THE PROEDURE"
  " BEING PARSED"
  "GLOBAL VARIABLES REFERENCED : HOLD_LINK_KEY / CURRENTWORD/"
  " CONSECCOMPARE/"
  "CALLING MODULES : PARSE_PROCEDURE/ PARSE_MAIN/"
VAR
  EXCESS_KEY, APPEND_KEY : HASHKEY;
  SEARCH, FOUND, DONE : BOOLEAN;
BEGIN
  EXCESS_KEY := HOLD_LINK_KEY;
  HASH_APPEND (APPEND_KEY, SEARCH, FOUND);
  IF FOUND
    THEN
      BEGIN
        IF CALLCOUNT = MAXPROCEDURECALLS
          THEN
              THEN
                APPEND_TABLE (APPEND_KEY, CALLCOUNT, EXCESS_KEY);
          END
        ELSE
          IF SEARCH
            THEN
              BEGIN
                APPEND_KEY := 0;
                DONE := FALSE;
                REPEAT
              END
          END
      END
  END
END
******************************************************************************
IF CURRENTWORD[j] = HASHTABLE[APPEND_KEY].PROCGAME
THEN
BEGIN
IF CALLCOUNT = MAXPROCEDURECALLS
THEN
IF CURRENTWORD[j] <> CONSEC_COMPARE[j]
THEN
ALLOCATE_MORE_SUBPHRASE(CALLCOUNT,EXCESS_KEY);
APPENDTABLE(APPEND_KEY, CALLCOUNT, EXCESS_KEY);
DONE := TRUE
END
ELSE
IF APPEND_KEY = MAXUNIQUEPROCEDURES
THEN DONE := TRUE
ELSE APPEND_KEY := APPEND_KEY + 1
UNTIL DONE
END
END

******************************************************************************
PROCEDURE HASH_ENTER (VAR KEY : HASHTYPE);
"PURPOSE : DETERMINE THE HASHTYPE FOR THE PROCEDURE BEING
"PARSED"
"GLOBAL VARIABLES REFERENCED : CURRENTWORD/ ENTER_COUNT/
"HASHTABLE / HALT_FLAG /
"NULL /
"CALLING MODULES : ENTERTABLE/ OUTPUT_CONTROL/
VAR
1. NEED_SLOT : 1..MAXWORDTYPE;
2. CHAR_INDEX : WORDINDEX;
BEGIN
IF ENTER_COUNT < MAXUNIQUEPROCEDURES - (MAXWORDTYPE - 1)
THEN
BEGIN
KEY := 1;
NEED_SLOT := 1;
FOR CHAR_INDEX := 1 TO MAXWORDLENGTH DO
IF CURRENTWORD[j] <> '
THEN KEY := KEY * (ORD(CURRENTWORD[j] + CHAR_INDEX)) MOD MAX1
MOD HASHTABLE;
FOR I := 1 TO MAXWORDTYPE DO
IF CURRENTWORD[j] <> '
THEN NEED_SLOT := I
ELSE I := MAXWORDTYPE;
IF CURRENTWORD[j] <> HASHTABLE[KEY].PROCGAME
THEN
CASE NEED_SLOT OF
1 : WHILE (HASHTABLE[KEY].NODENUMBER <> NULL) AND
(CURRENTWORD[j] <> HASHTABLE[KEY].PROCGAME) DO
KEY := KEY * 11 MOD MAX11;
2 : WHILE (HASHTABLE[KEY + 1].NODENUMBER <> NULL) OR
(CURRENTWORD[j] <> HASHTABLE[KEY].PROCGAME) DO

E-19
KEY := (KEY + 1) MOD HASHMAX1;
3 : WHILE ((HASHTABLE[KEY],NODENUMBER <> NULL) OR
(HASHTABLE[KEY + 1],NODENUMBER <> NULL) OR
(HASHTABLE[KEY + 2],NODENUMBER <> NULL) AND
(CURRENTWORD <= HASHTABLE[KEY],PROCNAME)) DO
KEY := (KEY + 1) MOD HASHMAX1;
4 : WHILE ((HASHTABLE[KEY],NODENUMBER <> NULL) OR
(HASHTABLE[KEY + 1],NODENUMBER <> NULL) OR
(HASHTABLE[KEY + 2],NODENUMBER <> NULL) OR
(CURRENTWORD <= HASHTABLE[KEY + 1],PROCNAME)) DO
KEY := (KEY + 1) MOD HASHMAX1
END
ELSE
BEGIN
HALT_FLAG := TRUE;
EXIT_PROCESSES
END;
ENTER_COUNT := ENTER_COUNT + NEED_SLOT
END;

******************************************************************************************
PROCEDURE ENCTHTABLE (VAR CURRENTNODE : SMALLINTEGER);
"PURPOSE : TO ENTER PROCEDURE INFORMATION INTO THE HASHTABLE
"GLOBAL VARIABLES REFERENCED : HOLD_LINK_KEY/ HOLD_HASHKEY/
" NULL / CURRENTWORD/
" " CURRENTLINE /
" " CALLING MODULES : PARSE_PROCEDURE/ PARSE_MAIN/
" " VAR
734 NEXT_HASHKEY, HASH_KEY : HASHKEY;
735 I : 1..MAXWORDTYPE;
736 BEGIN
737 IF NOT HALT_FLAG
738 THEN
739 BEGIN
740 HASH_ENTER (HASH_KEY);
741 WITH HASHTABLE[HASH_KEY] DO
742 BEGIN
743 IF NODENUMBER = NULL
744 THEN NODENUMBER := CURRENTNODE
745 ELSE CURRENTNODE := CURRENTNODE + 1;
746 PROCNAME := CURRENTWORD;1;
747 PROC_LINENO := CURRENTLINE
748 END;
749 NEXT_HASHKEY := HASH_KEY;
750 FOR I := 2 TO MAXWORDTYPE DO
751 IF CURRENTWORD <= 13
752 THEN
753 BEGIN
754 NEXT_HASHKEY := NEXT_HASHKEY + 1;
755 HASHTABLE[NEXT_HASHKEY],PROCNAME := CURRENTWORD;
E-15
HASHTABLE[NEXT_HASHKEY].NODENUMBER := 0
END
ELSE I := MAXWORDTYPE;

HOLD_HASHKEY := HASH_KEY;
HOLD_LINK_KEY := HASH_KEY
END

***************************************************************************************

PROCEDURE PARSE_PROCEDURE (VAR CURRENTNODE : SHORTINTEGER;)
VAR GETNEXTWORD : BOOLEAN;

"PURPOSE : TO PARSE A PROCEDURE, EXAMINING IT FOR

"SUBPROCEDURE CALLS

"GLOBAL VARIABLES REFERENCED : CURRENTWORD/ CONSEC_COMPARE/

"HALT_FLAG /

"CALLING MODULES : BUILDTABLE/


VAR CALLCOUNT, STACK : SHORTINTEGER:
BEGIN
GETNEXTWORD := TRUE;
GETWORD:
IF (CURRENTWORD[1] = 'U') OR
(CURRENTWORD[2] = 'O') OR
(CURRENTWORD[3] = 'T') OR
(CURRENTWORD[4] = 'U') OR
(CURRENTWORD[5] = 'S') OR
(CURRENTWORD[6] = 'T') OR
(CURRENTWORD[7] = 'D')
THEN
BEGIN
CURRENTNODE := CURRENTNODE + 1;
ENTERTABLE (CURRENTNODE)
END:
CALLCOUNT := 0;
CONSEC_COMPARE[1] := 'DUMMYDUMMYDUMMYDUMMY';
GETWORD:
IF (CURRENTWORD[1] = PROCEDURE) OR
FUNCTION
THEN GETNEXTWORD := FALSE
ELSE
IF (CURRENTWORD[1] = FORWARD) AND
EXTERNAL
THEN
BEGIN
STACK := 11
WHILE (CURRENTWORD[1] = BEGIN) AND NOT (EOF) := 0
GETWORD:
REPEAT
GETWORD;
IF (CURRENTWORD13 = BEGIN_) OR
(CURRENTWORD13 = CASE_)
THEN STACK := STACK + 1
ELSE
IF CURRENTWORD13 = END_
THEN STACK := STACK - 1
ELSE
IF NOT HALT_FLAG
THEN APPEND_DETERMINE (CALLCOUNT)
UNTIL (STACK = 0) OR (EOF)
END;
IF EOF
THEN
BEGIN
HALT_FLAG := TRUE;
UNBALANCED_ERROR (END_STRING)
END
END;

**********************************************************************************
PROCEDURE PARSE_MAIN (VAR CURRENTNODE : SHORTINTEGER);

"PURPOSE : TO PARSE THE MAIN PROGRAM, EXAMINING IT FOR"
"PROCEDURE CALLS"
"GLOBAL VARIABLES REFERENCED : CURRENTWORD/ CONSEC_COMPARE/
"HALT_FLAG /
"CALLING MODULES : BUILTABLE/

VAR
CALLCOUNT, STACK : SHORTINTEGER;
BEGIN
CURRENTWORD13 := MAIN_; 
CURRENTNODE := CURRENTNODE + 1;
ENTERTABLE (CURRENTNODE);
CALLCOUNT := 0;
CONSEC_COMPARE13 := MAIN_1
STACK := 1;
REPEAT
GETWORD;
IF (CURRENTWORD13 = BEGIN_) OR
(CURRENTWORD13 = CASE_)
THEN STACK := STACK + 1
ELSE
IF CURRENTWORD13 = END_
THEN STACK := STACK - 1
ELSE
IF NOT HALT_FLAG
THEN APPEND_DETERMINE (CALLCOUNT)
UNTIL (STACK = 0) OR (EOF);
IF EOF
THEN
BEGIN
HALT_FLAG := TRUE;
UNBALANCED_ERROR (END_STRING)
END
END
PGMEND

******************************************************************************
PROCEDURE PARSER_FUNCTION (VAR GET_NEXTWORD : BOOLEAN);
  "PURPOSE: TO PARSE A FUNCTION, EXAMINING IT TO DETERMINE
  WHERE THE FUNCTION TERMINATES"
  "GLOBAL VARIABLES REFERENCED: CURRENTWORD/"
  "CALLING MODULES: BUILDTABLE/"
  VAR
    CALLCOUNT, STACK : SHORTINTEGER;
BEGIN
  CALLCOUNT := 0;
  REPEAT
    GETWORD;
    UNTIL (CURRENTWORD[1] = PROCEDURE_) OR
    (CURRENTWORD[1] = BEGIN_ OR (EOF));
  IF CURRENTWORD[1] = BEGIN_
  THEN
    BEGIN
      STACK := 1;
      REPEAT
        GETWORD;
        IF (CURRENTWORD[1] = BEGIN_) OR
        (CURRENTWORD[1] = CASE_)
        THEN STACK := STACK + 1
        ELSE
        IF CURRENTWORD[1] = END_
        THEN STACK := STACK - 1
        UNTIL (STACK = 0) OR (EOF);
  END
  ELSE GET_NEXTWORD := FALSE;
  IF EOF
  THEN
    BEGIN
      HALT_FLAG := TRUE;
      UNBALANCED_ERROR (END_STRING)
    END
  END;

******************************************************************************
PROCEDURE BUILDTABLE;
  "PURPOSE: TO CONTROL THE PARSING OF PROCEDURES, FUNCTIONS, AND THE MAIN PROGRAM"
  "GLOBAL VARIABLES REFERENCED: CURRENTWORD/ PROCEDUREFLAG/"
  "CALLING MODULES: MAIN/"
  VAR
    GET_NEXTWORD : BOOLEAN;
    CURRENTNODE : SHORTINTEGER;
BEGIN
  CURRENTNODE := 0;
  "E-18"
GETNEXTWORD := TRUE;
WHILE NOT EOF AND NOT HALT_FLAG DO
  BEGIN
    IF GETNEXTWORD
      THEN GETWORD:
    IF CURRENTWORD[1] = FUNCTION_
      THEN PARSE_FUNCTION (GETNEXTWORD)
    ELSE IF CURRENTWORD[1] = PROCEDURE_
      THEN
        BEGIN
          PROCEDUREFLAG := TRUE;
          PARSE_PROCEDURE (CURRENTNODE; GETNEXTWORD)
        END
    ELSE
      IF CURRENTWORD[1] = BEGIN_
        THEN
          BEGIN
            MAIN_FLAG := TRUE;
            PARSE_MAIN (CURRENTNODE)
          END
      END:
  END:

************************************************************************************************************************
PROCEDURE TRAVERSE_TABLE(HASHKEY : HASHKEY; VAR STDO : SHORTINTEGER;
  VAR SPACER : SHORTINTEGER) FORWARD;
************************************************************************************************************************

PROCEDURE TRAVERSE_TABLE:
  "PURPOSE : TO TRAVERSE THE HASHTABLE, OUTPUTTING THE
  "INFORMATION BUILT BY BUILDTABLE
  "GLOBAL VARIABLES REFERENCED : EXCESS_CALL_FLAG/ CALL_FLAG/
  "HASHTABLE / JUSTIFY /
  "CALLING MODULES : OUTPUT_CONTROL/ TRAVERSE_TABLE (RECURSIVE)/

VAR
  A, I, X, Y, Z : SHORTINTEGER;
  XHASHKEY, NEXT_HASHKEY, LONG_NAME_KEY : HASHKEY;
  REPEATITIVE_FLAG : BOOLEAN;
BEGIN
  WITHHASHTABLE(HASH_KEY) DO
  BEGIN
    IF NO_SUBPROC_CALLS > 0
    THEN
      BEGIN
        XHASHKEY := HASH_KEY;
        A := 01;
        CALL_FLAG := TRUE;
      FOR X := 1 TO NO_SUBPROC_CALLS DO
        BEGIN
          EXCESS_CALL_FLAG := FALSE;
          A := A + 1;
          IF (X = MAXCALLS) OR (X = MAXCALLS * 2 - 1) OR
            E-19
(x = MAXCALLS1 * 3 - 2) OR (x = MAXCALLS1 * 4 - 3) OR
(x = MAXCALLS1 * 5 - 4) OR (x = MAXCALLS1 * 6 - 5) OR
(x = MAXCALLS1 * 7 - 6) OR (x = MAXCALLS1 * 8 - 7)
THEN
BEGIN
XHASHKEY := HASHTABLE[XHASHKEY].LINK_KEY;
A := 1
END;
IF x = MAXCALLS1 * 9 - 8
THEN
BEGIN
EXCESS_PROCEDURES;
X := NO_SUBPROC_CALLS + 1;
EXCESS_CALL_FLAG := TRUE;
END;
IF NOT EXCESS_CALL_FLAG
THEN
BEGIN
SEQNO := SEQNO + 1;
NEXT_HASHKEY := HASHTABLE[XHASHKEY].NESTEDCAJ.SUBPROC_HASHKEY;
JUSTIFY := FALSE;
WRITEINT (SEQNO * 4);
WRITEINT (HASHTABLE[NEXT_HASHKEY].PROC_LINE.Number + 61);
FOR Y := 1 TO SPACER DO
WRITESTRING (' ' (10:1));
JUSTIFY := TRUE;
WRITEINT (HASHTABLE[XHASHKEY].NESTEDCAJ.SUBPROC_LINE.Number + 61);
WRITENAME (HASHTABLE[NEXT_HASHKEY].PROCNAME, SPACER);
LONG_NAME_KEY := NEXT_HASHKEY
IF LONG_NAME_KEY <= MAXUNIQUEPROCEDURES - (MAXWORDTYPE - 1)
THEN
FOR i := 2 TO MAXWORDTYPE DO
BEGIN
LONG_NAME_KEY := LONG_NAME_KEY + 1;
IF (HASHTABLE[LONG_NAME_KEY].NODENUMBER = 0)
THEN WRITENAME (HASHTABLE[LONG_NAME_KEY].PROCNAME, SPACER)
ELSE I := MAXWORDTYPE;
END;
BEGIN
IF HASHTABLE[XHASHKEY].NESTEDCAJ.SUBPROC_CONSLCALLS > 0
THEN
BEGIN
WRITESTRING (' ' (10:1));
JUSTIFY := FALSE;
Z := HASHTABLE[XHASHKEY].NESTEDCAJ.SUBPROC_CONSLCALLS + 1;
WRITEINT (Z * 2);
WRITESTRING (' ' (10:1));
END;
REPEITIVE_FLAG := TRUE;
IF HASHTABLE[NEXT_HASHKEY].REPEITIVE > 0
THEN
BEGIN
REPEITIVE_FLAG := FALSE;
IF HASHTABLE[NEXT_HASHKEY].NO_SUBPROC_CALLS > 0
THEN
E-20
BEGIN
   JUSTIFY := FALSE;
   WRITEINT (* {10}*)
   WRITEINT (HASHTABLE[HEADER].REPEITIVE, NL, 5);
   WRITEINT (* {10}*)
   WRITE (NL)
END
ELSE
BEGIN
   WRITEINT (* {10}*)
   WRITE (NL)
END
END:
IF REPEITIVE_FLAG
THEN
BEGIN
   HASHTABLE[NEXT_HASHKEY].REPEITIVE := SLGW;
   SPACER := SPACER + 1;
   WRITEINT (* {10}*)
   WRITE (NL)
   IF SPACER < 20
   THEN TRAVERSE_TABLE(NEXT_HASHKEY, SEQUO, SPACER)
   ELSE
      BEGIN
         EXCESS_NESTING :=
         X := NO_SUBPROC_CALLS + 1;
         SPACER := SPACER - 1
      END
END
END:
BEGIN
   SPACER := SPACER - 1
END
END:

"********************************************************************************

PROCEDURE OUTPUT_CONTROL:
"PURPOSE : TO CONTROL THE PRINTING OF THE OUTPUT
"GLOBAL VARIABLES REFERENCED : HOLD_HASHKEY/ CURRENTWORD/
"" HASHTABLE / MAIN_FLAG /
"JUSTIFY /
"" CALLING MODULES : MAIN/

VAR
   X, SEQUO, SPACER : SHORTINTEGER;
BEGIN
   HEADER:
   SEQUO := 1;
   SPACER := 0;
   IF NOT MAIN_FLAG
   THEN
      BEGIN
         E-21
CURRENTWORD[i] := HASHTABLE[RESULT][PROCNAME := MAIN];
HASHTABLE[RESULT][PROCNAME := MAIN];
BEGIN
WITH HASHTABLE[RESULT][PROCNAME := MAIN] DO
BEGIN
  NODENUMBER := NULL - 1;
  PROCNAME := CURRENTWORD[i];
  PROC_LINENO := HASHTABLE[RESULT][PROCNAME][PROC_LINENO];
END;
WITH HASHTABLE[RESULT][PROCNAME := MAIN] DO
BEGIN
  PROC_LINENO := PROC_LINENO + 1;
  FOR x := 1 TO NO_SUBPROC_CALLS DO
    IF NEXTDECODED[SUBPROC_HASHKEY] = RESULT THEN
      NEXTDECODED[SUBPROC_HASHKEY] := RESULT
END;
END;
WRITEINT (SEQNO,4);
WRITEINT (HASHTABLE[RESULT][PROCNAME][PROC_LINENO],6);
WRITEINT (* MAIN(0) *
WRITE (NL);
THVERSE_TABLE (RESULT, SEQNO, SPACER);
SEQNO := SEQNO + 1;
JUSTIFY := FALSE;
WRITEINT (SEQNO,4);
CURRENTLINE := CURRENTLINE - 1;
WRITEINT (CURRENTLINE,6);
WRITESTRING (* END(0) *
WRITE (NL);
END;

******************************************************************************
PROCEDURE INITIALIZE;
*** PURPOSE : TO INITIALIZE THE GLOBAL VARIABLES AND
*** ALL HASHTABLE SLOTS
*** GLOBAL VARIABLES REFERENCED : EXCESS_CALL_FLAG/ CURRENTCHAR/
*** HOLD_LINK_KEY / CURRENTWORD/
*** HOLD_HASHKEY / HASHTABLE/
*** PROCEDURE_FLAG / ENTER_COUNT/
*** CONSEC_COMPARE / NULL/
*** CALL_FLAG / MAIN_FLAG/
*** NUMBERS / LETTERS/
*** HURT_FLAG / JUSTIFY/
*** CALLING MODULES : MAIN/
VAR
  X, Y : SHORTINTEGER;
BEGIN
  WRITESTRING (* SEQUENTIAL PASCAL PROCEDURAL HIERARCHY GENERATOR(101) *
  WRITE (NL);
  NULL := 32767;
  CURRENTLINE := 1;
  ENTER_COUNT := 0;
END;
BEGIN
  NODENUMBER := NULL;
  FOR Y := 1 TO MAXWORDLENGTH DO
    PROCNAMECY] := ' ';
  REPETITIVE := 0;
  PROC_LINENO := 0;
  NO_SUBPROC_CALLS := 0;
  FOR Y := 1 TO MAXPROCEDURECALLS DO
    NESTEDCYJ, SUBPROC_CONSECALLS := 0;
    NESTEDCYJ, SUBPROC_LINENO := 0;
    NESTEDCYJ, SUBPROC_HASHKEY := 0;
  END
END

BEGIN
  FOR X := 1 TO MAXWORDTYPE DO
    FOR Y := 1 TO MAXWORDLENGTH DO
      BEGIN
        CURRENTWORD[X, Y] := ' ';
        CONSEC_COMPARE[X, Y] := ' ';
      END
END

BEGIN
  INITIALIZE;
  BUILDTABLE;
  IF NOT HALT_FLAG THEN
    BEGIN
      IF PROCEDUREFLAG THEN OUTPUT_CONTROL ELSE THAILEN;
      IF PROCEDUREFLAG AND NOT CALL_FLAG
      THEN
        E=23
THEN TRAILER
ELSE WRITE (EM)
END
END.
A PROCEDURE HIERARCHY GENERATOR FOR PASCAL

by

KENNETH D. HARMON

B. A., Pittsburg State University, 1967

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1980
A PROCEDURE HIERARCHY GENERATOR FOR PASCAL

This Master's Report presents a software engineering tool for the generation of the hierarchies of procedures in sequential PASCAL computer programs.

Programming in the PASCAL language is characterized by extensive use of physically small procedures. This structuring technique serves to keep each procedure relatively simple and understandable. However, a side effect of this technique is the difficulty it causes individuals who are attempting to gain a general knowledge of a large program or are attempting to manually trace execution in a large program. The pieces of large PASCAL programs may be easily understood, but when the pieces are combined to create a large program, the overall result can be quite complicated and interleaved.

To lessen the impact of this design characteristic, a procedure hierarchy generator has been designed and implemented. The generator functions as a stand-alone software package. Primarily, it is designed to output a hierarchical display of the procedure invoking structure of PASCAL programs. Users of this generator will be able to:

- Readily ascertain the structure of programs under investigation.
- Eliminate the requirement for extensive hand-written records during manual execution traces of programs under investigation.

- Additionally, use the output to verify that user programs being created are actually coded as designed (as pertains to the procedure invoking structure).

The procedure hierarchy generator adds a powerful software engineering tool to the KSU software package library. Users employing this tool can significantly increase their understanding of large PASCAL programs and, simultaneously, decrease the time and effort required to do so.