

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

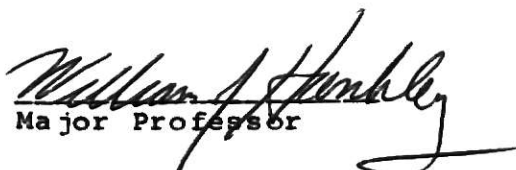
MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
MANHATTAN, KANSAS

1980

Approved by:


Major Professor

Spec. Coll.
LID
2668
.R4
1980
H37
c.2

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
II.	USER INFORMATION.....	7
	Executing the Generator.....	7
	Reading and Understanding.....	7
	the Generator Output	
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

A PROCEDURE HIERARCHY GENERATOR FOR PASCAL

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.

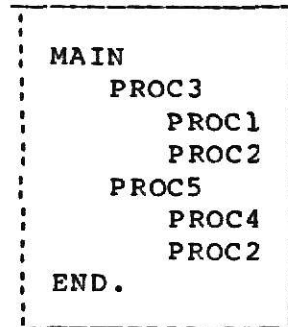


Figure I.1 Sample Non-Graphical Display

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 similiar 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--Hierarchy plus Input-Process-Output (HIPO) [FRE76]. 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 similiar--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:

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

- o Input programs must be free of all compilation errors to insure 100% reliability of generator output.
- o 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.
- o Procedure parameters are disregarded during output formulation.
- o Procedures must be explicitly called in the source program to appear in the output.
- o 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 WRITESTRING indicates that the INITIALIZE procedure, at line 1129, calls a subprocedure WRITESTRING which has been declared at program line 198.

5. Examine output line 11. 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).

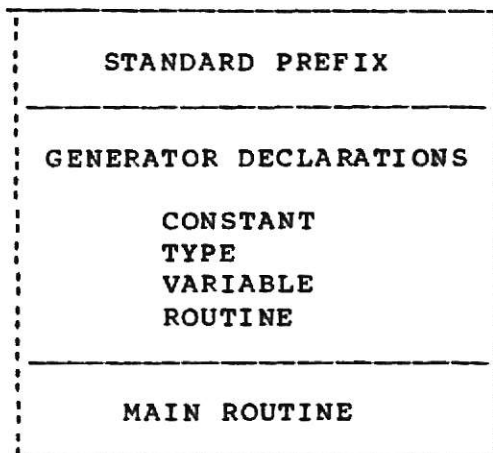


Figure III.1 Generator Program Structure

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.

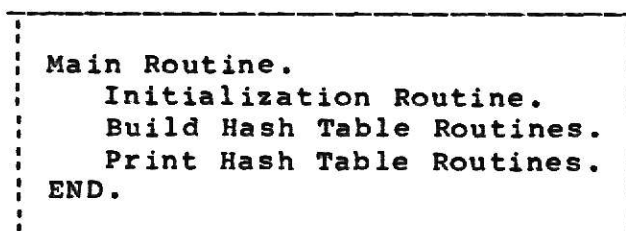


Figure III.2 Generator Routines Structure

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.
  WRITESTRING
  HEADER
    WRITESTRING
  WRITENAME
    WRITESTRING
  EXCESS_PROCEEDURES
    WRITESTRING
  EXCESS_NESTING
    WRITESTRING
  WRITEINT
    WRITESTRING
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

Column					
1	2	3	4	5	6
Seq No	Ln No of the Main Program BEGIN or END Instruction	"MAIN" or "END."			
Seq No	Called Proc Declaration Line Number	a.	Called Proc Name	optional b. or optional c.	optional c.
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:

- o procedure names <= 20 characters in length.
- o 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.

Similiarly, 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:

- o increase the number of characters allowed per slot per procedure name.
- o increase the number of subprocedure calls allowed per slot per procedure.
- o 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.

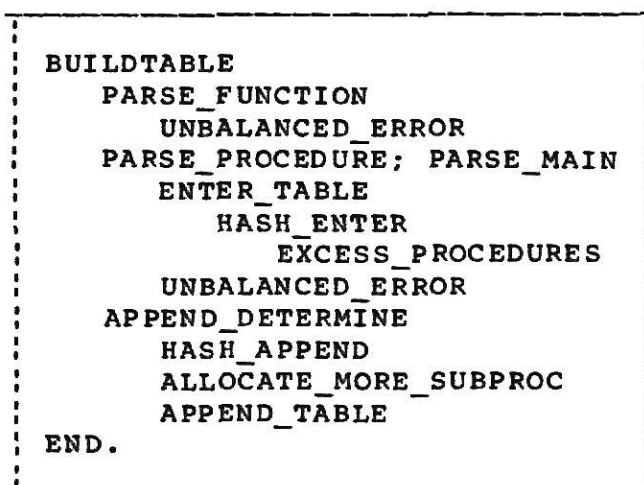


Figure III.5 Structure of the
Build Hash Table Routines

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_PROCEEDURES 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.

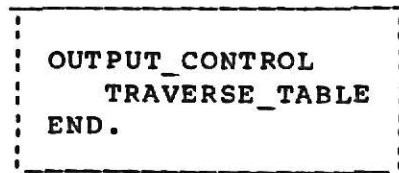


Figure III.6 Structure of the
Print Hash Table Routines

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. **TRAVERSE_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. **TRAVERSE_TABLE Routine.** The **TRAVERSE_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_PROCEEDURES 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 similiar 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:

```
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
  X1(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 "X1" instead of "X". The generator will check to see if "X1" 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_PROCEEDURES 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 similiar to output produced for procedures and the main routine.

IV. BIBLIOGRAPHY

- [BRI77] Brinch Hansen, Per, "The Architecture of Concurrent Programs", Prentice-Hall, Inc., 1977.
- [FRE76] Freeman, Peter, and Wasserman, Anthony I., "Tutorial on Software Design Techniques", IEEE Catalog No. 76CH1145-2C, 1976, pp. 181-188.
- [KAR60] Karp, R.M., "A Note on the Application of Graph Theory to Digital Computer Programming", Inform. Contr., Vol. 3, 1960, pp. 179-190.
- [PAI77] Paige, Michael R., "On Partitioning Program Graphs", IEEE Transactions on Software Engineering", Vol. SE-3, No. 6, Nov 1977, pp. 386-393.

PROCEDURE HIERARCHY GENERATOR OUTPUT LISTING

SEQUENTIAL PASCAL PROCEDURE HIERARCHY GENERATOR

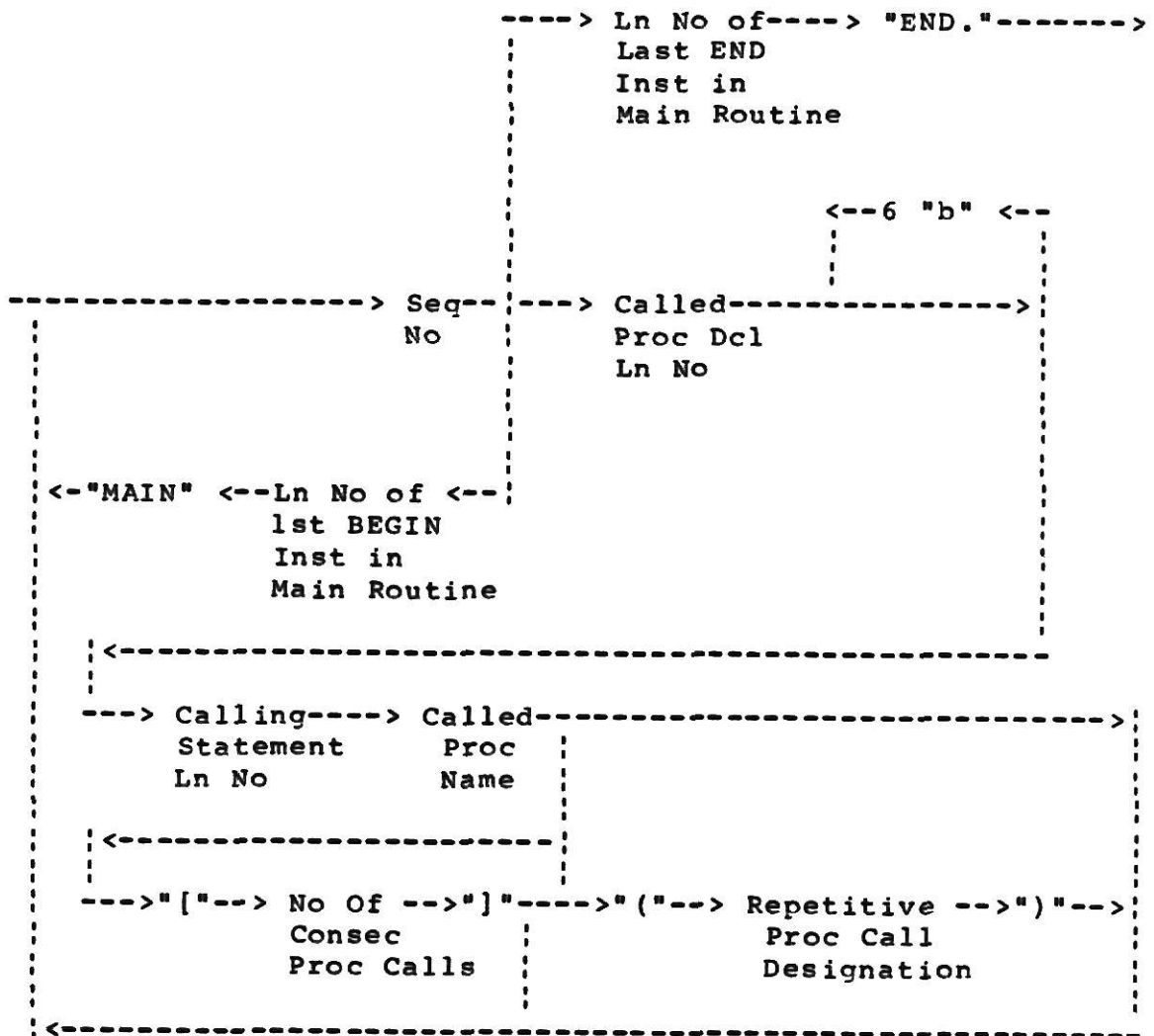
WARNING : IF YOUR INPUT PROGRAM HAS COMPILATION
***** ERRORS IN PASS1 THROUGH PASS5, THE
***** FOLLOWING OUTPUT CANNOT BE CONSIDERED
***** RELIABLE.

```
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
8      355          478 GETCHAR
9      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_PROCEEDURES
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)
```

42	82		277 WRITE [2]
43	461	792	GETWORD [3] (7)
44	617	819	APPEND_DETERMINE
45	581	630	HASH_APPEND
46	557	638	ALLOCATE_MORE_SUBPROC
47	528	639	APPENDTABLE
48	557	655	ALLOCATE_MORE_SUBPROC
49	528	656	APPENDTABLE
50	375	826	UNBALANCED_ERROR (12)
51	831	937	PARSE_MAIN
52	725	843	ENTERTABLE (30)
53	461	848	GETWORD (7)
54	617	857	APPEND_DETERMINE (44)
55	375	863	UNBALANCED_ERROR (12)
56	1063	1185	OUTPUT_CONTROL
57	302	1074	HEADER
58	82	308	WRITE
59	198	309	WRITESTRING (3)
60	82	310	WRITE
61	198	311	WRITESTRING (3)
62	82	312	WRITE
63	198	313	WRITESTRING (3)
64	82	314	WRITE
65	198	315	WRITESTRING (3)
66	82	316	WRITE [2]
67	668	1082	HASH_ENTER (31)
68	218	1097	WRITEINT [2]
69	82	238	WRITE [5]
70	198	1099	WRITESTRING (3)
71	82	1100	WRITE
72	947	1101	TRAVERSE_TABLE
73	251	983	EXCESS_PROCEDURES (32)
74	218	993	WRITEINT [2] (68)
75	198	996	WRITESTRING (3)
76	218	998	WRITEINT (68)
77	173	999	WRITENAME [2]
78	82	191	WRITE
79	198	1013	WRITESTRING (3)
80	218	1016	WRITEINT (68)
81	198	1017	WRITESTRING [2] (3)
82	218	1029	WRITEINT (68)
83	198	1030	WRITESTRING (3)
84	82	1031	WRITE
85	198	1035	WRITESTRING (3)
86	82	1036	WRITE
87	198	1044	WRITESTRING (3)
88	82	1045	WRITE
89	947	1047	TRAVERSE_TABLE (72)
90	283	1050	EXCESS_NESTING
91	198	291	WRITESTRING (3)
92	82	292	WRITE
93	198	293	WRITESTRING (3)
94	82	294	WRITE
95	198	295	WRITESTRING (3)

96	82		296	WRITE
97	198		297	WRITESTRING (3)
98	82		298	WRITE
99	218	1104	WRITEINT [2] (68)	
100	198	1107	WRITESTRING (3)	
101	82	1108	WRITE	
102	321 1186	TRAILER [2]		
103	198	331	WRITESTRING (3)	
104	82	332	WRITE	
105	198	333	WRITESTRING [2] (3)	
106	82	336	WRITE [2]	
107	82 1189	WRITE		
108	1191	END.		

OUTPUT SYNTAX DIAGRAM



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.

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

Node No	Proc Name	Link Key	Repet- itive Ind	Proc Dcl Ln No	No of Subproc Calls	No of Consec Calls	LnNo of Subproc Call	Subproc Key
9999	20(X)	999	999	9999	999	999	9999	999

Translation Key -

Node : Node creation sequence number.
No

Proc : Procedure name.
Name

Link : Index to continuation nodes.
Key

Repet- : Repetitive procedure expansion indicator.
itive
Ind

Proc : Procedure declaration line number.
Dcl
Ln No

Subproc : Number of subprocedure calls.
Calls

No of : Number of consecutive subprocedure calls.
Consec
Calls

LnNo of : Program line number where subprocedure was
Subproc called.
Call

Subproc : Index to subprocedure declaration node.
Key

9 : Numeric character.

X : Alphanumeric character.

PROCEDURE HIERARCHY GENERATOR SOURCE LISTING

```

1
2 "PER BRINCH HANSEN
3
4 INFORMATION SCIENCE
5 CALIFORNIA INSTITUTE OF TECHNOLOGY
6
7 UTILITY PROGRAMS FOR
8 THE SOLO SYSTEM
9
10 18 MAY 1975"
11
12 "*****
13 # PREFIX #
14 *****"
15
16
17 CONST NL = '(:10:)'; FF = '(:12:)'; CR = '(:13:)';
18 EM = '(:25:)';
19
20 CONST PAGELENGTH = 512;
21 TYPE PAGE = ARRAY (1..PAGELENGTH.) OF CHAR;
22
23 CONST LINELENGTH = 132;
24 TYPE LINE = ARRAY (1..LINELENGTH.) OF CHAR;
25
26 CONST IDLENGTH = 12;
27 TYPE IDENTIFIER = ARRAY (1..IDLENGTH.) OF CHAR;
28
29 TYPE FILE = 1..2;
30
31 TYPE FILEKIND = (EMPTY, SCRATCH, ASCII, SEQCODE, CONCODE);
32
33 TYPE FILEATTR = RECORD
34     KIND: FILEKIND;
35     ADDR: INTEGER;
36     PROTECTED: BOOLEAN;
37     NOTUSED: ARRAY (1..5.) OF INTEGER
38 END;
39
40 TYPE IODEVICE =
41     (TYPEDEVICE, DISKDEVICE, TAPEDEVICE, PRINTDEVICE, CARDDDEVICE);
42
43 TYPE IOOPERATION = (INPUT, OUTPUT, MOVE, CONTROL);
44
45 TYPE IOARG = (WRITEEOF, REWIND, UPSPACE, BACKSPACE);
46
47 TYPE IORESULT =
48     (COMPLETE, INTERVENTION, TRANSMISSION, FAILURE,
49     ENDFILE, ENDMEDIUM, STARTMEDIUM);
50
51 TYPE IOPARAM = RECORD
52     OPERATION: IOOPERATION;
53     STATUS: IORESULT;

```

```

54         ARG: IOARG
55     END;
56
57     TYPE TASKKIND = (INPUTTASK, JOBTASK, OUTPUTTASK);
58
59     TYPE ARGTAG =
60         (NILTYPE, BOOLTYPE, INTTYPE, IDTYPE, PTRTYPE);
61
62     TYPE POINTER = @BOOLEAN;
63
64     TYPE ARGTYPE = RECORD
65         CASE TAG: ARGTAG OF
66             NILTYPE, BOOLTYPE: (BOOL: BOOLEAN);
67             INTTYPE: (INT: INTEGER);
68             IDTYPE: (ID: IDENTIFIER);
69             PTRTYPE: (PTR: POINTER)
70         END;
71
72     CONST MAXARG = 10;
73     TYPE ARGLIST = ARRAY (1..MAXARG.) OF ARGTYPE;
74
75     TYPE ARGSEQ = (INP, OUT);
76
77     TYPE PROGRESRESULT =
78         (TERMINATED, OVERFLOW, POINTERERROR, RANGEERROR, VARIANTERROR,
79          HEAPLIMIT, STACKLIMIT, CODELIMIT, TIMELIMIT, CALLERERROR);
80
81     PROCEDURE READ(VAR C: CHAR);
82     PROCEDURE WRITE(C: CHAR);
83
84     PROCEDURE OPEN(F: FILE; ID: IDENTIFIER; VAR FOUND: BOOLEAN);
85     PROCEDURE CLOSE(F: FILE);
86     PROCEDURE GET(F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);
87     PROCEDURE PUT(F: FILE; P: INTEGER; VAR BLOCK: UNIV PAGE);
88     FUNCTION LENGTH(F: FILE): INTEGER;
89
90     PROCEDURE MARK(VAR TOP: INTEGER);
91     PROCEDURE RELEASE(TOP: INTEGER);
92
93     PROCEDURE IDENTIFY(HEADER: LINE);
94     PROCEDURE ACCEPT(VAR C: CHAR);
95     PROCEDURE DISPLAY(C: CHAR);
96
97
98     PROGRAM GEN;
99
100
101     "*****
102     *          GENERATOR CONSTANT, TYPE, AND VARIABLE DECLARATIONS          *
103     *****"
104
105     CONST
106         MAXWORDLENGTH = 20;
107         MAXSTRINGLENGTH = 60;

```



```

108     MAXUNIQUEPROCEDURES = 200;
109     HASHMAX1 = 201;
110     MAXPROCEDURECALLS = 20;
111     MAXCALLS1 = 21;
112     SPAN = 27;
113     MAXWORDTYPE = 4;
114     END_STRING = '(:0:)' ;
115     QUOTEMARK = '(:34:)' ;
116     APOSTROPHE = '(:39:)' ;
117     LEFT_PAREN = '(:40:)' ;
118     RIGHT_PAREN = '(:41:)' ;
119     LEFT_BRACKET = '(:91:)' ;
120     RIGHT_BRACKET = '(:93:)' ;
121     LEFT_BRACE = '(:123:)' ;
122     RIGHT_BRACE = '(:125:)' ;
123     BEGIN_ = 'BEGIN' ;
124     END_ = 'END' ;
125     FUNCTION_ = 'FUNCTION' ;
126     PROCEDURE_ = 'PROCEDURE' ;
127     MAIN_ = 'MAIN' ;
128     VAR_ = 'VAR' ;
129     FORWARD_ = 'FORWARD' ;
130     TYPE_ = 'TYPE' ;
131     CONST_ = 'CONST' ;
132     CASE_ = 'CASE' ;
133     EXTERN_ = 'EXTERN' ;
134
135     TYPE
136     LINES = ARRAY[1..MAXSTRINGLENGTH] OF CHAR;
137     WORDINDEX = 1..MAXWORDLENGTH;
138     WORD = ARRAY[WORDINDEX] OF CHAR;
139     WORDTYPE = ARRAY[1..MAXWORDTYPE] OF WORD;
140     HASHKEY = 0..MAXUNIQUEPROCEDURES;
141     SUBPROC = ARRAY[1..MAXPROCEDURECALLS] OF
142         RECORD
143             SUBPROC_CONSECALLS : SHORTINTEGER;
144             SUBPROC_LINENO : SHORTINTEGER;
145             SUBPROC_HASHKEY : SHORTINTEGER
146         END;
147     HASH_TABLE = ARRAY[HASHKEY] OF
148         RECORD
149             NODENUMBER : SHORTINTEGER;
150             PROCNAME : WORD;
151             LINK_KEY : SHORTINTEGER;
152             REPETITIVE : SHORTINTEGER;
153             PROC_LINENO : SHORTINTEGER;
154             NO_SUBPROC_CALLS : SHORTINTEGER;
155             NESTED : SUBPROC
156         END;
157
158     VAR
159     NULL, ENTER_COUNT, CURRENTLINE : SHORTINTEGER;
160     EXCESS_CALL_FLAG, MAIN_FLAG, JUSTIFY, CALLFLAG,
161     PROCEDUREFLAG, HALT_FLAG : BOOLEAN;

```

```

162 HOLD_LINK_KEY, HOLD_HASHKEY : HASHKEY;
163 LETTERS, NUMBERS, LETTERS_NUMBERS : SET OF CHAR;
164 CURRENTCHAR : CHAR;
165 HASHTABLE : HASH_TABLE;
166 CURRENTWORD, CONSEC_COMPARE : WORDTYPE;
167
168 "*****
169 * GENERATOR PROCEDURE DECLARATIONS *
170 *****"
171
172 "*****"
173 PROCEDURE WRITENAME (C : WORD; SPACER : SHORTINTEGER);
174 "PURPOSE : WRITES A PROCEDURE NAME TO LOGICAL UNIT 2 "
175 "GLOBAL VARIABLES REFERENCED : NONE. "
176 "CALLING MODULES : TRAVERSE_TABLE "
177
178 VAR
179 I : WORDINDEX;
180 PR_COUNT_CHAR : SHORTINTEGER;
181 BEGIN
182 I := 1;
183 PR_COUNT_CHAR := SPACER * 6 + 18;
184 WHILE I <= MAXWORDLENGTH DO
185 BEGIN
186 IF C(I) <> ' '
187 THEN
188 BEGIN
189 PR_COUNT_CHAR := PR_COUNT_CHAR + 1;
190 IF PR_COUNT_CHAR < 133
191 THEN WRITE (C(I))
192 END;
193 I := I + 1
194 END
195 END;
196
197 "*****"
198 PROCEDURE WRITESTRING (TEXT : LINES);
199 "PURPOSE : WRITES A CHARACTER STRING TO LOGICAL UNIT 2 "
200 "GLOBAL VARIABLES REFERENCED : NONE. "
201 "CALLING MODULES : EXCESS_NESTING / OUTPUT_CONTROL/ "
202 " EXCESS_PROCEURES / HEADER / "
203 " TRAVERSE_TABLE / TRAILER / "
204 " UNBALANCED_ERROR / INITIALIZE / "
205
206 VAR
207 I : SHORTINTEGER;
208 BEGIN
209 I := 1;
210 WHILE TEXT(I) <> END_STRING DO
211 BEGIN
212 WRITE (TEXT(I));
213 I := I+1
214 END
215 END;

```

```

216
217 "*****"
218 PROCEDURE WRITEINT (N, DIGITS : INTEGER);
219 "PURPOSE : WRITES INTEGERS TO LOGICAL UNIT 2 "
220 "GLOBAL VARIABLES REFERENCED : JUSTIFY "
221 "CALLING MODULES : TRAVERSE_TABLE/ OUTPUT_CONTROL/ "
222
223 VAR
224 REM, DIG, I : 1..6;
225 NUM : ARRAY[1..6] OF CHAR;
226 BEGIN
227 REM := N;
228 DIG := 1;
229 REPEAT
230 NUM[DIG] := CHR(ABS(REM) MOD 10 + ORD('0'));
231 REM := REM DIV 10;
232 DIG := SUCC(DIG)
233 UNTIL REM = 0;
234 IF NOT JUSTIFY
235 THEN
236 FOR I := DIGITS DOWNT0 1 DO
237 IF I >= DIG
238 THEN WRITE (' ')
239 ELSE WRITE (NUM[I])
240 ELSE
241 BEGIN
242 WRITE (' ');
243 FOR I := DIG - 1 DOWNT0 1 DO
244 WRITE (NUM[I]);
245 FOR I := DIG TO DIGITS DO
246 WRITE (' ')
247 END
248 END;
249
250 "*****"
251 PROCEDURE EXCESS_PROCEEDURES;
252 "PURPOSE : OUTPUT USER MESSAGE IF THE PROGRAM BEING PARSED "
253 " HAS TOO MANY PROCEDURES FOR THE HASHTABLE TO STORE "
254 " OR THE PROCEDURE BEING PARSED HAS TOO MANY "
255 " SUBPROCEDURES FOR THE HASHTABLE TO STORE "
256 "GLOBAL VARIABLES REFERENCED : HALT_FLAG "
257 "CALLING MODULES : TRAVERSE_TABLE / HASH_ENTER / "
258
259 BEGIN
260 IF HALT_FLAG
261 THEN WRITESTRING (' THE PROGRAM(:0:))')
262 ELSE WRITESTRING (' THE PROCEDURE(:0:))');
263 WRITESTRING (' BEING ANALYZED HAS TOO MANY(:0:))');
264 WRITE (NL);
265 IF HALT_FLAG
266 THEN WRITESTRING (' PROCEDURES(:0:))')
267 ELSE WRITESTRING (' SUBPROCEDURE CALLS(:0:))');
268 WRITESTRING (' FOR THE GENERATOR, AS(:0:))');
269 WRITE (NL);

```

```

270      WRITESTRING (' CURRENTLY SET UP. CONTACT OPERATIONS TO(:0:)):
271      WRITE (NL);
272      WRITESTRING (' REQUEST THAT THIS ARBITRARY LIMIT BE RAISED.( :0:)):
273      WRITE (NL);
274      IF HALT_FLAG
275      THEN WRITESTRING (' NO GENERATOR OUTPUT CAN BE PROVIDED.( :0:)):
276      ELSE WRITESTRING (' THIS PROCEDURE WILL NOT BE ANALYZED FURTHER.( :0:)):
277      WRITE (NL);
278      IF HALT_FLAG
279      THEN WRITE (EM)
280      END;
281
282      "*****"
283      PROCEDURE EXCESS_NESTING:
284      "PURPOSE : OUTPUT USER MESSAGE THAT THE PROCEDURE BEING PARSED      "
285      "      HAS A NESTING LEVEL SO DEEP THAT IT CANNOT BE              "
286      "      DISPLAYED ON A PRINTED PAGE                                "
287      "GLOBAL VARIABLES REFERENCED : NONE.                              "
288      "CALLING MODULES : TRAVERSE_TABLE/"
289
290      BEGIN
291      WRITESTRING(' THE PROCEDURE CURRENTLY BEING EXPANDED HAS A(:0:)):
292      WRITE(NL);
293      WRITESTRING(' NESTING LEVEL DEEPER THAN 19. THIS EXCEEDS(:0:)):
294      WRITE(NL);
295      WRITESTRING(' THE PHYSICAL PAPER PRINTOUT LIMITATIONS. THE(:0:)):
296      WRITE(NL);
297      WRITESTRING(' PROCEDURE CALLS WILL NOT BE EXPANDED FURTHER.( :0:)):
298      WRITE(NL);
299      END;
300
301      "*****"
302      PROCEDURE HEADER:
303      "PURPOSE : WRITES OUT A HEADER MESSAGE PRIOR TO OUTPUTTING DATA  "
304      "GLOBAL VARIABLES REFERENCED : NONE.                              "
305      "CALLING MODULES : OUTPUT-CONTROL/"
306
307      BEGIN
308      WRITE (NL);
309      WRITESTRING ('          WARNING : IF YOUR INPUT PROGRAM HAS COMPILATION(:0:)):
310      WRITE (NL);
311      WRITESTRING ('          ***** ERRORS IN PASS1 THROUGH PASS5, THE(:0:)):
312      WRITE (NL);
313      WRITESTRING ('          ***** FOLLOWING OUTPUT CANNOT BE CONSIDERED(:0:)):
314      WRITE (NL);
315      WRITESTRING ('          ***** RELIABLE.( :0:)):
316      WRITE (NL);
317      WRITE (NL);
318      END;
319
320      "*****"
321      PROCEDURE TRAILER:
322      "PURPOSE : WRITES OUT A TRAILER MESSAGE SUBSEQUENT TO              "
323      "      OUTPUTTING DATA, IF REQUIRED                                "

```

```

324 "GLOBAL VARIABLES REFERENCED : PROCEDUREFLAG/
325 "CALLING MODULES : MAIN/
326
327 BEGIN
328   IF PROCEDUREFLAG
329     THEN
330       BEGIN
331         WRITESTRING ('***** PROGRAM CONTAINS PROCEDURES: HOWEVER, *****(:0:)');
332         WRITE (NL);
333         WRITESTRING ('      MAIN DOES NOT INVOKE ANY OF THEM.(:0:)')
334       END
335     ELSE WRITESTRING ('***** PROGRAM CONTAINS NO PROCEDURES. *****(:0:)');
336     WRITE (NL);
337     WRITE (EM)
338   END;
339
340 "*****"
341 FUNCTION EOF : BOOLEAN;
342
343   BEGIN
344     EOF := CURRENTCHAR = EM
345   END;
346
347 "*****"
348 FUNCTION EOLN : BOOLEAN;
349
350   BEGIN
351     EOLN := CURRENTCHAR = NL
352   END;
353
354 "*****"
355 PROCEDURE GETCHAR;
356   "PURPOSE : READS A SINGLE CHARACTER FROM THE INPUT FILE
357   "GLOBAL VARIABLES REFERENCED : CURRENTCHAR/ CURRENTLINE/
358   "CALLING MODULES : SKIP_COMMENT_APOSTROPHE / GETWORD /
359   "      SKIP_PAREN_BRACKET_BRACE /
360
361   BEGIN
362     IF EOF
363       THEN CURRENTCHAR := EM
364     ELSE
365       IF EOLN
366         THEN
367           BEGIN
368             CURRENTCHAR := ' ';
369             CURRENTLINE := CURRENTLINE + 1
370           END
371         ELSE READ (CURRENTCHAR)
372     END;
373
374 "*****"
375 PROCEDURE UNBALANCED_ERROR (CHAR1 : CHAR);
376   "PURPOSE : PRINTS OUT AN ERROR MESSAGE TO USER IF THE INPUT
377   "      PROGRAM CONTAINS ANY UNBALANCED QUOTATION MARKS.

```

```

378      "          APOSTROPHES, PARENTHESES, BRACKETS, BRACES, OR      "
379      "          BEGIN-END INSTRUCTIONS.                                "
380      "          KEEPS THE GENERATOR FROM CRASHING AS A RESULT OF      "
381      "          I/O ERRORS.                                            "
382      "GLOBAL VARIABLES REFERENCED : NONE.                             "
383      "CALLING MODULES : SKIP_COMMENT_APOSTROPHE / PARSE_MAIN /        "
384      "                  PARSE_PROCEDURE / PARSE_FUNCTION/            "
385      "                  SKIP-PAREN_BRACKET_BRACE/                    "
386
387      BEGIN
388      WRITESTRING ( 'INPUT PROGRAM CONTAINS AN UNBALANCED SET OF(:0:)');
389      CASE CHAR1 OF
390      QUOTEMARK      : WRITESTRING (' QUOTATION MARKS.(:0:)');
391      APOSTROPHE     : WRITESTRING (' APOSTROPHES.(:0:)');
392      LEFT_PAREN     : WRITESTRING (' PARENTHESIS.(:0:)');
393      LEFT_BRACKET   : WRITESTRING (' BRACKETS.(:0:)');
394      LEFT_BRACE     : WRITESTRING (' BRACES.(:0:)');
395      END_STRING     : WRITESTRING (' BEGIN-ENDS.(:0:)')
396      END;
397      WRITE (NL);
398      WRITESTRING ('CORRECT AND RESUBMIT YOUR PROGRAM FOR PROCESSING.(:0:)');
399      WRITE (NL);
400      WRITE (EM);
401      END;
402
403      "*****
404      PROCEDURE SKIP_COMMENT_APOSTROPHE (CHAR1 : CHAR);
405      "PURPOSE : SKIPS OVER CHARACTERS ENCLOSED IN QUOTEMARKS      "
406      "          AND APOSTROPHES                                     "
407      "GLOBAL VARIABLES REFERENCED : CURRENTCHAR/                  "
408      "CALLING MODULES : GETWORD/                                    "
409
410      BEGIN
411      IF CURRENTCHAR <> CHAR1
412      THEN
413      REPEAT
414      GETCHAR;
415      IF EOF
416      THEN
417      BEGIN
418      HALT_FLAG := TRUE;
419      UNBALANCED_ERROR (CHAR1)
420      END
421      UNTIL (CURRENTCHAR = CHAR1) OR (EOF);
422      IF NOT HALT_FLAG
423      THEN GETCHAR
424      END;
425
426      "*****
427      PROCEDURE SKIP_PAREN_BRACKET_BRACE (CHAR1, CHAR2 : CHAR);
428      "PURPOSE : SKIPS OVER CHARACTERS ENCLOSED IN PARENTHESIS,    "
429      "          BRACKETS, AND BRACES                                 "
430      "GLOBAL VARIABLES REFERENCED : CURRENTCHAR/                  "
431      "CALLING MODULES : GETWORD/                                    "

```

```

432
433   VAR
434   I : SHORTINTEGER;
435   BEGIN
436   I := 1;
437   REPEAT
438   GETCHAR;
439   IF EOF
440   THEN
441   BEGIN
442   HALT_FLAG := TRUE;
443   UNBALANCED_ERROR (CHAR1)
444   END;
445   IF CURRENTCHAR = APOSTROPHE
446   THEN
447   BEGIN
448   GETCHAR;
449   WHILE (CURRENTCHAR <> APOSTROPHE) AND NOT (EOF) DO
450   GETCHAR
451   END;
452   IF CURRENTCHAR = CHAR1
453   THEN I := I + 1
454   ELSE
455   IF CURRENTCHAR = CHAR2
456   THEN I := I - 1
457   UNTIL (I = 0) OR (HALT_FLAG)
458   END;
459
460   "*****"
461   PROCEDURE GETWORD;
462   "PURPOSE : READS A SINGLE WORD FROM THE INPUT FILE"
463   "GLOBAL VARIABLES REFERENCED : CURRENTCHAR/ LETTERS"
464   "                                NUMBERS / LETTERS_NUMBERS/"
465   "                                CURRENTWORD/"
466   "CALLING MODULES : PARSE_PROEDURE/ PARSE_MAIN/"
467   "                  PARSE_FUNCTION / BUILDTABLE/"
468
469   VAR
470   INDEX, BLANKINDEX : 0..MAXWORDLENGTH;
471   I, X : SHORTINTEGER;
472   CHAR1, CHAR2 : CHAR;
473   BEGIN
474   LETTERS_NUMBERS := LETTERS - NUMBERS;
475   WHILE NOT (EOF OR (CURRENTCHAR IN LETTERS_NUMBERS)) DO
476   CASE CURRENTCHAR OF
477   QUOTEMARK : BEGIN
478   GETCHAR;
479   SKIP_COMMENT_APOSTROPHE (QUOTEMARK)
480   END;
481   APOSTROPHE : BEGIN
482   GETCHAR;
483   SKIP_COMMENT_APOSTROPHE (APOSTROPHE);
484   END;
485   LEFT_PAREN : BEGIN

```

```

486             CHAR1 := LEFT_PAREN;
487             CHAR2 := RIGHT_PAREN;
488             SKIP_PAREN_BRACKET_BRACE (CHAR1, CHAR2);
489         END;
490     LEFT_BRACKET : BEGIN
491         CHAR1 := LEFT_BRACKET;
492         CHAR2 := RIGHT_BRACKET;
493         SKIP_PAREN_BRACKET_BRACE (CHAR1, CHAR2);
494     END;
495     LEFT_BRACE : BEGIN
496         CHAR1 := LEFT_BRACE;
497         CHAR2 := RIGHT_BRACE;
498         SKIP_PAREN_BRACKET_BRACE (CHAR1, CHAR2);
499     END;
500     ELSE : GETCHAR
501     END;
502     I := 0;
503     WHILE CURRENTCHAR IN LETTERS_NUMBERS DO
504     BEGIN
505         LETTERS_NUMBERS := LETTERS + NUMBERS;
506         INDEX := 0;
507         I := I + 1;
508         WHILE (CURRENTCHAR IN LETTERS_NUMBERS) AND
509             (INDEX <> MAXWORDLENGTH) DO
510         BEGIN
511             INDEX := INDEX + 1;
512             CURRENTWORD[I,INDEX] := CURRENTCHAR;
513             GETCHAR
514         END;
515         IF INDEX < MAXWORDLENGTH
516         THEN
517             FOR BLANKINDEX := INDEX + 1 TO MAXWORDLENGTH DO
518                 CURRENTWORD[I,BLANKINDEX] := ' ';
519             END;
520         IF I < MAXWORDTYPE
521         THEN
522             FOR X := I + 1 TO MAXWORDTYPE DO
523                 FOR BLANKINDEX := 1 TO MAXWORDLENGTH DO
524                     CURRENTWORD[X,BLANKINDEX] := ' ';
525             END;
526         *****
527     PROCEDURE APPENDTABLE (APPEND_KEY : HASHKEY; VAR CALLCOUNT :
528     SHORTINTEGER; EXCESS_KEY : HASHKEY);
529     "PURPOSE : APPEND INFORMATION ON SUBPROCEDURES CALLED BY
530     " THE PROCEDURE BEING PARSED
531     "GLOBAL VARIABLES REFERENCED : CURRENTWORD / CURRENTCHAR/
532     " HASHTABLE / CURRENTLINE/
533     " HOLD_LINK_KEY/
534     "CALLING MODULES : APPEND_DETERMINE/
535     BEGIN
536     IF CURRENTWORD[I] <> CONSEC_COMPARE[I]
537     THEN

```



```

540 BEGIN
541 CALLCOUNT := CALLCOUNT + 1;
542 WITH HASHTABLE[EXCESS_KEY].NESTED[CALLCOUNT] DO
543 BEGIN
544 SUBPROC_LINENO := CURRENTLINE;
545 SUBPROC_HASHKEY := APPEND_KEY
546 END;
547 WITH HASHTABLE[HOLD_HASHKEY] DO
548 NO_SUBPROC_CALLS := NO_SUBPROC_CALLS + 1;
549 CONSEC_COMPARE := CURRENTWORD
550 END
551 ELSE
552 WITH HASHTABLE[EXCESS_KEY].NESTED[CALLCOUNT] DO
553 SUBPROC_CONSECALLS := SUBPROC_CONSECALLS + 1
554 END;
555
556 "*****"
557 PROCEDURE ALLOCATE_MORE_SUBPROC (VAR CALLCOUNT : SHORTINTEGER;
558 VAR EXCESS_KEY : HASHKEY);
559 "PURPOSE : OBTAIN AN ADDITIONAL HASHTABLE ENTRY POINT FOR "
560 " MORE SUBPROCEDURE CALL INFORMATION "
561 "GLOBAL VARIABLES REFERENCED : HOLD_LINK_KEY/ NULL/ "
562 " HASHTABLE / "
563 "CALLING MODULES : APPEND_DETERMINE/ "
564
565 BEGIN
566 CALLCOUNT := 0;
567 IF HOLD_LINK_KEY = MAXUNIQUEPROCEDURES
568 THEN EXCESS_KEY := 0
569 ELSE EXCESS_KEY := HOLD_LINK_KEY + 1;
570 WHILE HASHTABLE[EXCESS_KEY].NODENUMBER <> NULL DO
571 IF EXCESS_KEY = MAXUNIQUEPROCEDURES
572 THEN EXCESS_KEY := 0
573 ELSE EXCESS_KEY := EXCESS_KEY + 1;
574 WITH HASHTABLE[EXCESS_KEY] DO
575 NODENUMBER := NODENUMBER + 1;
576 HASHTABLE[HOLD_LINK_KEY].LINK_KEY := EXCESS_KEY;
577 HOLD_LINK_KEY := EXCESS_KEY
578 END;
579
580 "*****"
581 PROCEDURE HASH_APPEND (VAR KEY : HASHKEY; VAR SEARCH : BOOLEAN;
582 VAR FOUND : BOOLEAN);
583 "PURPOSE : DETERMINE THE HASHKEY FOR THE SUBPROCEDURE "
584 " CALLED BY THE PROCEDURE BEING PARSED "
585 "GLOBAL VARIABLES REFERENCED : CURRENTWORD/ HASHTABLE/ "
586 "CALLING MODULES : APPEND_DETERMINE/ "
587
588 VAR
589 CHAR_INDEX : WORDINDEX;
590 BEGIN
591 KEY := 1;
592 FOR CHAR_INDEX := 1 TO MAXWORDLENGTH DO
593 IF CURRENTWORD[CHAR_INDEX] <> ' '

```

```

594         THEN KEY := KEY * (ORD(CURRENTWORD[1,CHAR_INDEX]) MOD SPAN)
595             MOD HASHMAX1;
596     IF HASHTABLE[KEY],NODENUMBER = NULL
597     THEN
598         BEGIN
599             SEARCH := FALSE;
600             FOUND := FALSE
601         END
602     ELSE
603         IF HASHTABLE[KEY].PROCNAME = CURRENTWORD[1]
604         THEN
605             BEGIN
606                 FOUND := TRUE;
607                 SEARCH := FALSE
608             END
609         ELSE
610             BEGIN
611                 SEARCH := TRUE;
612                 FOUND := FALSE
613             END
614     END;
615
616 *****
617 PROCEDURE APPEND_DETERMINE (VAR CALLCOUNT : SHORTINTEGER);
618 "PURPOSE : DETERMINE IF THE CURRENTWORD IS A PROCEDURE      "
619 "          CALL WHICH SHOULD BE APPENDED TO THE PROCEDURE   "
620 "          BEING PARSED                                       "
621 "GLOBAL VARIABLES REFERENCED : HOLD_LINK_KEY / CURRENTWORD/  "
622 "          CONSEC_COMPARE/                                     "
623 "CALLING MODULES : PARSE_PROCEDURE/ PARSE_MAIN/              "
624
625 VAR
626     EXCESS_KEY, APPEND_KEY : HASHKEY;
627     SEARCH, FOUND, DONE : BOOLEAN;
628 BEGIN
629     EXCESS_KEY := HOLD_LINK_KEY;
630     HASH_APPEND (APPEND_KEY, SEARCH, FOUND);
631     IF FOUND
632     THEN
633         BEGIN
634             IF CALLCOUNT = MAXPROCEDURECALLS
635             THEN
636                 IF CURRENTWORD[1] <> CONSEC_COMPARE[1]
637                 THEN
638                     ALLOCATE_MORE_SUBPROC (CALLCOUNT, EXCESS_KEY);
639                     APPENDTABLE (APPEND_KEY, CALLCOUNT, EXCESS_KEY)
640                 END
641             ELSE
642                 IF SEARCH
643                 THEN
644                     BEGIN
645                         APPEND_KEY := 0;
646                         DONE := FALSE;
647                         REPEAT

```

```

648         IF CURRENTWORD[1] = HASHTABLE[APPEND_KEY].PROCNAME
649         THEN
650             BEGIN
651                 IF CALLCOUNT = MAXPROCEDURECALLS
652                 THEN
653                     IF CURRENTWORD[1] <> CONSEC_COMPARE[1]
654                     THEN
655                         ALLOCATE_MORE_SUBPROC(CALLCOUNT, EXCESS_PLY);
656                         APPENDTABLE (APPEND_KEY, CALLCOUNT, EXCESS_KEY);
657                         DONE := TRUE
658                     END
659                 ELSE
660                     IF APPEND_KEY = MAXUNIQUEPROCEDURES
661                     THEN DONE := TRUE
662                     ELSE APPEND_KEY := APPEND_KEY + 1
663                 UNTIL DONE
664             END
665         END;
666
667 *****
668 PROCEDURE HASH_ENTER (VAR KEY : HASHKEY);
669     "PURPOSE : DETERMINE THE HASHKEY FOR THE PROCEDURE BEING      "
670     "   PARSED                                                    "
671     "GLOBAL VARIABLES REFERENCED : CURRENTWORD/ ENTER_COUNT/      "
672     "                               HASHTABLE / HALT_FLAG /         "
673     "                               NULL /                          "
674     "CALLING MODULES : ENTERTABLE/ OUTPUT_CONTROL/               "
675
676     VAR
677         I, NEED_SLOT : 1..MAXWORDTYPE;
678         CHAR_INDEX : WORDINDEX;
679     BEGIN
680         IF ENTER_COUNT < MAXUNIQUEPROCEDURES - (MAXWORDTYPE - 1)
681         THEN
682             BEGIN
683                 KEY := 1;
684                 NEED_SLOT := 1;
685                 FOR CHAR_INDEX := 1 TO MAXWORDLENGTH DO
686                     IF CURRENTWORD[1,CHAR_INDEX] <> ' '
687                     THEN KEY := (KEY * (ORD(CURRENTWORD[1,CHAR_INDEX]) MOD SPAN)
688                               MOD HASHMAX1;
689                 FOR I := 2 TO MAXWORDTYPE DO
690                     IF CURRENTWORD[1,I] <> ' '
691                     THEN NEED_SLOT := I
692                     ELSE I := MAXWORDTYPE;
693                 IF CURRENTWORD[1] <> HASHTABLE[KEY].PROCNAME
694                 THEN
695                     CASE NEED_SLOT OF
696                     1 : WHILE ((HASHTABLE[KEY].NODENUMBER <> NULL) AND
697                               (CURRENTWORD[1] <> HASHTABLE[KEY].PROCNAME)) DO
698                         KEY := (KEY + 1) MOD HASHMAX1;
699                     2 : WHILE ((HASHTABLE[KEY].NODENUMBER <> NULL) OR
700                               (HASHTABLE[KEY + 1].NODENUMBER <> NULL)) AND
701                               (CURRENTWORD[1] <> HASHTABLE[KEY].PROCNAME)) DO

```

```

702         KEY := (KEY + 1) MOD HASHMAX1;
703     3 : WHILE ((HASHTABLE[KEY].NODENUMBER <> NULL) OR
704             ((HASHTABLE[KEY + 1].NODENUMBER <> NULL) OR
705             (HASHTABLE[KEY + 2].NODENUMBER <> NULL)) AND
706             (CURRENTWORD[1] <> HASHTABLE[KEY].PROCNAME)) DO
707         KEY := (KEY + 1) MOD HASHMAX1;
708     4 : WHILE ((HASHTABLE[KEY].NODENUMBER <> NULL) OR
709             ((HASHTABLE[KEY + 1].NODENUMBER <> NULL) OR
710             (HASHTABLE[KEY + 2].NODENUMBER <> NULL) OR
711             (HASHTABLE[KEY + 3].NODENUMBER <> NULL)) AND
712             (CURRENTWORD[1] <> HASHTABLE[KEY].PROCNAME)) DO
713         KEY := (KEY + 1) MOD HASHMAX1
714     END
715 END
716 ELSE
717 BEGIN
718     HALT_FLAG := TRUE;
719     EXCESS_PROCEEDURES
720 END;
721 ENTER_COUNT := ENTER_COUNT + NEED_SLOT
722 END;
723
724 "*****"
725 PROCEDURE ENTERTABLE (VAR CURRENTNODE : SHORTINTEGRAL);
726 "PURPOSE : TO ENTER PROCEDURE INFORMATION INTO THE HASHTABLE      "
727 "GLOBAL VARIABLES REFERENCED : HOLD_LINK_KEY/ HOLD_HASHKEY/      "
728 "                                HALT_FLAG / HASHTABLE /          "
729 "                                NULL / CURRENTWORD /            "
730 "                                CURRENTLINE /                    "
731 "CALLING MODULES : PARSE_PROCEDURE/ PARSE_MAIN/                  "
732
733 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_LINEEND := CURRENTLINE
748                 END;
749                 NEXT_HASHKEY := HASH_KEY;
750                 FOR I := 2 TO MAXWORDTYPE DO
751                     IF CURRENTWORD[I,1] <> ' '
752                     THEN
753                         BEGIN
754                             NEXT_HASHKEY := NEXT_HASHKEY + 1;
755                             HASHTABLE[NEXT_HASHKEY].PROCNAME := CURRENTWORD[I];

```

```

756             HASHTABLE[NEXT_HASHKEY].NODENUMBER := 0
757         END
758         ELSE I := MAXWORDTYPE;
759         HOLD_HASHKEY := HASH_KEY;
760         HOLD_LINK_KEY := HASH_KEY
761     END
762 END;
763
764 *****
765 PROCEDURE PARSE_PROCEDURE (VAR CURRENTNODE : SHORTINTEGER;
766                             VAR GETNEXTWORD : BOOLEAN);
767     "PURPOSE : TO PARSE A PROCEDURE, EXAMINING IT FOR      "
768     "      SUBPROCEDURE CALLS                                "
769     "GLOBAL VARIABLES REFERENCED : CURRENTWORD/ CONSEC_COMPARE/ "
770     "      HALT_FLAG /                                         "
771     "CALLING MODULES : BUILDTABLE/                             "
772
773     VAR
774         CALLCOUNT, STACK : SHORTINTEGER;
775     BEGIN
776         GETNEXTWORD := TRUE;
777         GETWORD;
778         IF (CURRENTWORD[1,1] <> 'N') OR
779             (CURRENTWORD[1,2] <> 'O') OR
780             (CURRENTWORD[1,3] <> 'T') OR
781             (CURRENTWORD[1,4] <> 'U') OR
782             (CURRENTWORD[1,5] <> 'S') OR
783             (CURRENTWORD[1,6] <> 'E') OR
784             (CURRENTWORD[1,7] <> 'D')
785         THEN
786             BEGIN
787                 CURRENTNODE := CURRENTNODE + 1;
788                 ENTERTABLE (CURRENTNODE)
789             END;
790             CALLCOUNT := 0;
791             CONSEC_COMPARE[1] := 'DUMMYDUMMYDUMMYDUMMY';
792             GETWORD;
793             IF (CURRENTWORD[1] = PROCEDURE_) OR
794                 (CURRENTWORD[1] = FUNCTION_)
795             THEN GETNEXTWORD := FALSE
796             ELSE
797                 IF (CURRENTWORD[1] <> FORWARD_) AND
798                     (CURRENTWORD[1] <> EXTERN_)
799                 THEN
800                     IF (CURRENTWORD[1] = VAR_) OR
801                         (CURRENTWORD[1] = BEGIN_) OR
802                         (CURRENTWORD[1] = TYPE_) OR
803                         (CURRENTWORD[1] = CONST_)
804                     THEN
805                         BEGIN
806                             STACK := 1;
807                             WHILE (CURRENTWORD[1] <> BEGIN_) AND NOT (EOF) DO
808                                 GETWORD;
809                                 REPEAT

```

```

810             GETWORD;
811             IF (CURRENTWORD[1] = BEGIN_) OR
812                (CURRENTWORD[1] = CASE_)
813             THEN STACK := STACK + 1
814             ELSE
815                 IF CURRENTWORD[1] = END_
816                 THEN STACK := STACK - 1
817                 ELSE
818                     IF NOT HALT_FLAG
819                     THEN APPEND_DETERMINE (CALLCOUNT)
820             UNTIL (STACK = 0) OR (EOF)
821         END;
822     IF EOF
823     THEN
824         BEGIN
825             HALT_FLAG := TRUE;
826             UNBALANCED_ERROR (END_STRING)
827         END
828     END;
829
830 *****
831 PROCEDURE PARSE_MAIN (VAR CURRENTNODE : SHORTINTEGER);
832 "PURPOSE : TO PARSE THE MAIN PROGRAM, EXAMINING IT FOR      "
833 "      PROCEDURE CALLS                                     "
834 "GLOBAL VARIABLES REFERENCED : CURRENTWORD/ CONSEC_COMPARE/ "
835 "      HALT_FLAG /                                         "
836 "CALLING MODULES : BUILDTABLE/                             "
837
838 VAR
839     CALLCOUNT, STACK : SHORTINTEGER;
840 BEGIN
841     CURRENTWORD[1] := MAIN_;
842     CURRENTNODE := CURRENTNODE + 1;
843     ENTERTABLE (CURRENTNODE);
844     CALLCOUNT := 0;
845     CONSEC_COMPARE[1] := MAIN_;
846     STACK := 1;
847     REPEAT
848         GETWORD;
849         IF (CURRENTWORD[1] = BEGIN_) OR
850            (CURRENTWORD[1] = CASE_)
851         THEN STACK := STACK + 1
852         ELSE
853             IF CURRENTWORD[1] = END_
854             THEN STACK := STACK - 1
855             ELSE
856                 IF NOT HALT_FLAG
857                 THEN APPEND_DETERMINE (CALLCOUNT)
858     UNTIL (STACK = 0) OR (EOF);
859     IF EOF
860     THEN
861         BEGIN
862             HALT_FLAG := TRUE;
863             UNBALANCED_ERROR (END_STRING)

```

```

864         END
865     END;
866
867     "*****"
868     PROCEDURE PARSE_FUNCTION (VAR GETNEXTWORD : BOOLEAN);
869     "PURPOSE : TO PARSE A FUNCTION, EXAMINING IT TO DETERMINE      "
870     "          WHERE THE FUNCTION TERMINATES                        "
871     "GLOBAL VARIABLES REFERENCED : CURRENTWORD/                   "
872     "CALLING MODULES : BUILDTABLE/                                "
873
874     VAR
875         CALLCOUNT, STACK : SHORTINTEGER;
876     BEGIN
877         CALLCOUNT := 0;
878         REPEAT
879             GETWORD
880             UNTIL ((CURRENTWORD[1] = PROCEDURE_) OR
881                  (CURRENTWORD[1] = BEGIN_) OR (EOF));
882             IF CURRENTWORD[1] = BEGIN_
883                 THEN
884                     BEGIN
885                         STACK := 1;
886                         REPEAT
887                             GETWORD;
888                             IF (CURRENTWORD[1] = BEGIN_) OR
889                                (CURRENTWORD[1] = CASE_)
890                                 THEN STACK := STACK + 1
891                                 ELSE
892                                     IF CURRENTWORD[1] = END_
893                                         THEN STACK := STACK - 1
894                                     UNTIL (STACK = 0) OR (EOF);
895                         END
896                     ELSE GETNEXTWORD := FALSE;
897                 IF EOF
898                     THEN
899                         BEGIN
900                             HALT_FLAG := TRUE;
901                             UNBALANCED_ERROR (END_STRING)
902                         END
903                 END;
904
905     "*****"
906     PROCEDURE BUILDTABLE;
907     "PURPOSE : TO CONTROL THE PARSING OF PROCEDURES, FUNCTIONS,    "
908     "          AND THE MAIN PROGRAM                                  "
909     "GLOBAL VARIABLES REFERENCED : CURRENTWORD/ PROCEDUREFLAG/     "
910     "          HALT_FLAG / MAIN_FLAG /                             "
911     "CALLING MODULES : MAIN/                                         "
912
913     VAR
914         GETNEXTWORD : BOOLEAN;
915         CURRENTNODE : SHORTINTEGER;
916     BEGIN
917         CURRENTNODE := 0;

```

```

918     GETNEXTWORD := TRUE;
919     WHILE NOT EOF AND NOT HALT_FLAG DO
920     BEGIN
921         IF GETNEXTWORD
922         THEN GETWORD;
923         IF CURRENTWORD[1] = FUNCTION_
924         THEN PARSE_FUNCTION (GETNEXTWORD)
925         ELSE
926             IF CURRENTWORD[1] = PROCEDURE_
927             THEN
928                 BEGIN
929                     PROCEDUREFLAG := TRUE;
930                     PARSE_PROCEDURE (CURRENTNODE, GETNEXTWORD)
931                 END
932             ELSE
933                 IF CURRENTWORD[1] = BEGIN_
934                 THEN
935                     BEGIN
936                         MAIN_FLAG := TRUE;
937                         PARSE_MAIN (CURRENTNODE)
938                     END
939                 END
940     END;
941
942     "*****"
943     PROCEDURE TRAVERSE_TABLE (HASH_KEY : HASHKEY; VAR SEQNO : SHORTINTEGER;
944                             VAR SPACER : SHORTINTEGER); FORWARD;
945
946     "*****"
947     PROCEDURE TRAVERSE_TABLE;
948     "PURPOSE : TO TRAVERSE THE HASHTABLE, OUTPUTTING THE          "
949     "          INFORMATION BUILT BY BUILDTABLE                     "
950     "GLOBAL VARIABLES REFERENCED : EXCESS_CALL_FLAG/ CALLFLAG/    "
951     "          HASHTABLE      / JUSTIFY /                          "
952     "CALLING MODULES : OUTPUT_CONTROL/ TRAVERSE_TABLE (RECURSIVE)/ "
953
954     VAR
955     A, I, X, Y, Z : SHORTINTEGER;
956     XHASHKEY, NEXT_HASHKEY, LONG_NAME_KEY : HASHKEY;
957     REPETITIVE_FLAG : BOOLEAN;
958     BEGIN
959     WITH HASHTABLE[HASH_KEY] DO
960     BEGIN
961         IF NO_SUBPROC_CALLS > 0
962         THEN
963             BEGIN
964                 XHASHKEY := HASH_KEY;
965                 A := 0;
966                 CALLFLAG := TRUE;
967                 FOR X := 1 TO NO_SUBPROC_CALLS DO
968                 BEGIN
969                     EXCESS_CALL_FLAG := FALSE;
970                     A := A + 1;
971                     IF (X = MAXCALLS1) OR (X = MAXCALLS1 * 2 - 1) OR

```



```

972      (X = MAXCALLS1 * 3 - 2) OR (X = MAXCALLS1 * 4 - 3) OR
973      (X = MAXCALLS1 * 5 - 4) OR (X = MAXCALLS1 * 6 - 5) OR
974      (X = MAXCALLS1 * 7 - 6) OR (X = MAXCALLS1 * 8 - 7)
975      THEN
976      BEGIN
977          XHASHKEY := HASHTABLE[XHASHKEY].LINK_KEY;
978          A := 1
979      END;
980      IF X = MAXCALLS1 * 9 - 8
981      THEN
982      BEGIN
983          EXCESS_PROCEDURES;
984          X := NO_SUBPROC_CALLS + 1;
985          EXCESS_CALL_FLAG := TRUE
986      END;
987      IF NOT EXCESS_CALL_FLAG
988      THEN
989      BEGIN
990          SEQNO := SEQNO + 1;
991          NEXT_HASHKEY := HASHTABLE[XHASHKEY].NESTEDCAJ.SUBPROC_HASHKEY;
992          JUSTIFY := FALSE;
993          WRITEINT (SEQNO,4);
994          WRITEINT (HASHTABLE[NEXT_HASHKEY].PROC_LINENO,6);
995          FOR Y := 1 TO SPACER DO
996              WRITESTRING (' '(:0:));
997          JUSTIFY := TRUE;
998          WRITEINT (HASHTABLE[XHASHKEY].NESTEDCAJ.SUBPROC_LINENO,6);
999          WRITENAME (HASHTABLE[NEXT_HASHKEY].PROCNAME, SPACER);
1000          LONG_NAME_KEY := NEXT_HASHKEY;
1001          IF LONG_NAME_KEY <= MAXUNIQUEPROCEDURES - (MAXWORDTYPE - 1)
1002          THEN
1003              FOR I := 2 TO MAXWORDTYPE DO
1004                  BEGIN
1005                      LONG_NAME_KEY := LONG_NAME_KEY + 1;
1006                      IF (HASHTABLE[LONG_NAME_KEY].NODENUMBER = 0)
1007                      THEN WRITENAME (HASHTABLE[LONG_NAME_KEY].PROCNAME,SPACER)
1008                      ELSE I := MAXWORDTYPE
1009                  END;
1010          IF HASHTABLE[XHASHKEY].NESTEDCAJ.SUBPROC_CONSECALLS > 0
1011          THEN
1012              BEGIN
1013                  WRITESTRING (' '(:0:));
1014                  JUSTIFY := FALSE;
1015                  Z := HASHTABLE[XHASHKEY].NESTEDCAJ.SUBPROC_CONSECALLS + 1;
1016                  WRITEINT (Z,2);
1017                  WRITESTRING (' '(:0:));
1018              END;
1019          REPETITIVE_FLAG := TRUE;
1020          IF HASHTABLE[NEXT_HASHKEY].REPETITIVE > 0
1021          THEN
1022              BEGIN
1023                  REPETITIVE_FLAG := FALSE;
1024                  IF HASHTABLE[NEXT_HASHKEY].NO_SUBPROC_CALLS > 0
1025                  THEN

```

```

1026             BEGIN
1027             JUSTIFY := FALSE;
1028             WRITESTRING (' ([:0:]);
1029             WRITEINT (HASHTABLE[NEXT_HASHKEY].REPETITIVE,3);
1030             WRITESTRING (' ([:0:]);
1031             WRITE (NL)
1032             END
1033         ELSE
1034             BEGIN
1035             WRITESTRING (' ([:0:]);
1036             WRITE (NL)
1037             END
1038         END;
1039     IF REPETITIVE_FLAG
1040     THEN
1041         BEGIN
1042             HASHTABLE[NEXT_HASHKEY].REPETITIVE := SEQNO;
1043             SPACER := SPACER + 1;
1044             WRITESTRING (' ([:0:]);
1045             WRITE (NL);
1046             IF SPACER < 20
1047             THEN TRAVERSE_TABLE(NEXT_HASHKEY, SEQNO, SPACER)
1048             ELSE
1049                 BEGIN
1050                     EXCESS_NESTING;
1051                     X := NO_SUBPROC_CALLS + 1;
1052                     SPACER := SPACER - 1
1053                 END
1054             END
1055         END
1056     END
1057 END;
1058 SPACER := SPACER - 1
1059 END
1060 END;
1061
1062 *****
1063 PROCEDURE OUTPUT_CONTROL;
1064 "PURPOSE : TO CONTROL THE PRINTING OF THE OUTPUT "
1065 "GLOBAL VARIABLES REFERENCED : HOLD_HASHKEY/ CURRENTWORD/ "
1066 " HASHTABLE / MAIN_FLAG / "
1067 " JUSTIFY / "
1068 "CALLING MODULES : MAIN/
1069
1070 VAR
1071     X, SEQNO, SPACER : SHORTINTEGER;
1072     HASH_KEY : HASHKEY;
1073 BEGIN
1074     HEADER;
1075     SEQNO := 1;
1076     SPACER := 0;
1077     IF NOT MAIN_FLAG
1078     THEN
1079         BEGIN

```

```

1080      CURRENTWORD[1] := HASHTABLE[HOLD_HASHKEY].PROCNAME;
1081      HASHTABLE[HOLD_HASHKEY].PROCNAME := MAIN_;
1082      HASH_ENTER (HASH_KEY);
1083      WITH HASHTABLE[HASH_KEY] DO
1084      BEGIN
1085          NODENUMBER := NULL - 1;
1086          PROCNAME := CURRENTWORD[1];
1087          PROC_LINENO := HASHTABLE[HOLD_HASHKEY].PROC_LINENO
1088      END;
1089      WITH HASHTABLE[HOLD_HASHKEY] DO
1090      BEGIN
1091          PROC_LINENO := PROC_LINENO + 1;
1092          FOR X := 1 TO NO_SUBPROC_CALLS DO
1093              IF NESTED[X].SUBPROC_HASHKEY = HOLD_HASHKEY
1094              THEN NESTED[X].SUBPROC_HASHKEY := HASH_KEY
1095          END
1096      END;
1097      WRITEINT (SEQNO,4);
1098      WRITEINT (HASHTABLE[HOLD_HASHKEY].PROC_LINENO,6);
1099      WRITESTRING (' MAIN(:0:1)');
1100      WRITE (NL);
1101      TRAVERSE_TABLE (HOLD_HASHKEY,SEQNO, SPACER);
1102      SEQNO := SEQNO + 1;
1103      JUSTIFY := FALSE;
1104      WRITEINT (SEQNO,4);
1105      CURRENTLINE := CURRENTLINE - 1;
1106      WRITEINT (CURRENTLINE,6);
1107      WRITESTRING (' END.(:0:)');
1108      WRITE (NL)
1109  END;
1110
1111  "*****"
1112  PROCEDURE INITIALIZE;
1113      "PURPOSE : TO INITIALIZE THE GLOBAL VARIABLES AND
1114      "          ALL HASHTABLE SLOTS
1115      "GLOBAL VARIABLES REFERENCED : EXCESS_CALL_FLAG/ CURRENTCHAR/
1116      "                                LETTERS_NUMBERS / CURRENTLINE/
1117      "                                HOLD_LINK_KEY   / CURRENTWORD/
1118      "                                HOLD_HASH_KEY   / HASHTABLE /
1119      "                                PROCEDUREFLAG  / ENTERCOUNT /
1120      "                                CONSEC_COMPARE / NULL      /
1121      "                                CALLFLAG      / MAIN_FLAG  /
1122      "                                NUMBERS       / LETTERS    /
1123      "                                HALT_FLAG     / JUSTIFY    /
1124      "CALLING MODULES : MAIN/
1125
1126  VAR
1127      X, Y : SHORTINTEGER;
1128  BEGIN
1129      WRITESTRING ('          SEQUENTIAL PASCAL PROCEDURE HIERARCHY GENERATOR(:0:1)');
1130      WRITE (NL);
1131      NULL := 32767;
1132      CURRENTLINE := 1;
1133      ENTER_COUNT := 0;

```

```

1134     HOLD_LINK_KEY := 0;
1135     HOLD_HASHKEY := 0;
1136     CURRENTCHAR := ' ';
1137     PROCEDUREFLAG := FALSE;
1138     HALT_FLAG := FALSE;
1139     MAIN_FLAG := FALSE;
1140     CALLFLAG := FALSE;
1141     EXCESS_CALL_FLAG := FALSE;
1142     JUSTIFY := FALSE;
1143     LETTERS := ['A','B','C','D','E','F','G','H','I','J','K','L','M','N','O',
1144                'P','Q','R','S','T','U','V','W','X','Y','Z','_'];
1145     NUMBERS := ['0','1','2','3','4','5','6','7','8','9'];
1146     LETTERS_NUMBERS := [J];
1147     FOR X := 0 TO MAXUNIQUEPROCEDURES DO
1148         WITH HASHTABLE[X] DO
1149             BEGIN
1150                 NODENUMBER := NULL;
1151                 FOR Y := 1 TO MAXWORDLENGTH DO
1152                     PROCNAME[Y] := ' ';
1153                     LINK_KEY := 0;
1154                     REPETITIVE := 0;
1155                     PROC_LINENO := 0;
1156                     NO_SUBPROC_CALLS := 0;
1157                     FOR Y := 1 TO MAXPROCEDURECALLS DO
1158                         BEGIN
1159                             NESTED[Y].SUBPROC_CONSECALLS := 0;
1160                             NESTED[Y].SUBPROC_LINENO := 0;
1161                             NESTED[Y].SUBPROC_HASHKEY := 0
1162                         END
1163                     END;
1164                 FOR X := 1 TO MAXWORDTYPE DO
1165                     FOR Y := 1 TO MAXWORDLENGTH DO
1166                         BEGIN
1167                             CURRENTWORD[X,Y] := ' ';
1168                             CONSEC_COMPARE[X,Y] := ' '
1169                         END
1170                     END;
1171                 END;
1172                 *****
1173                 **      **
1174                 ** MAIN **
1175                 **      **
1176                 *****
1177             BEGIN
1178                 INITIALIZE;
1179                 BUILDTABLE;
1180                 IF NOT HALT_FLAG
1181                     THEN
1182                         BEGIN
1183                             IF PROCEDUREFLAG
1184                                 THEN OUTPUT_CONTROL
1185                             ELSE TRAILER;
1186                             IF PROCEDUREFLAG AND NOT CALLFLAG

```

```
1188          THEN TRAILER
1189          ELSE WRITE (EM)
1190      END
1191  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:

- o Readily ascertain the structure of programs under investigation.

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