1970

A One Pass Algorithm for Compiling ALGOL 68 Declarations

Victor B. Schneider

Report Number:
70-053
A ONE-PASS ALGORITHM FOR COMPILING
ALGOL 68 DECLARATIONS

Victor B. Schneider

CSD TR 53
October 1970
(Revised October, 1971)

Research for this report was supported by N.S.F. grant GJ-851
to the Purdue Research Foundation
**CONTENTS**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>II. Techniques Used in Translation</td>
<td>4</td>
</tr>
<tr>
<td>Representation of ALGOL 68 Structures and Data Types</td>
<td>4</td>
</tr>
<tr>
<td>in the Compiler</td>
<td>4</td>
</tr>
<tr>
<td>Routines Used in Manipulating Compiler List Structures</td>
<td>4</td>
</tr>
<tr>
<td>Intermediate Language Generated by the Compiler</td>
<td>8</td>
</tr>
<tr>
<td>Preprocessing Implied by Rules of the Grammar</td>
<td>9</td>
</tr>
<tr>
<td>Stack Mechanisms Used by Compiler Subroutines</td>
<td>10</td>
</tr>
<tr>
<td>Criteria for Selecting Rule Forms in a Translation</td>
<td>12</td>
</tr>
<tr>
<td>Grammar</td>
<td></td>
</tr>
<tr>
<td>III. Translation of Array and Structure Declarations</td>
<td>16</td>
</tr>
<tr>
<td>Preliminaries</td>
<td>16</td>
</tr>
<tr>
<td>Array Declarations</td>
<td>17</td>
</tr>
<tr>
<td>Structure Declarations</td>
<td>21</td>
</tr>
<tr>
<td>IV. Translation of Mode Declarations</td>
<td>23</td>
</tr>
<tr>
<td>A Final Example</td>
<td>24</td>
</tr>
<tr>
<td>Bibliography</td>
<td>26</td>
</tr>
<tr>
<td>References Cited in Text</td>
<td>26</td>
</tr>
<tr>
<td>APPENDIX 1</td>
<td>27</td>
</tr>
<tr>
<td>A Partial Translation Grammar for ALGOL 68</td>
<td>27</td>
</tr>
<tr>
<td>APPENDIX 2</td>
<td>37</td>
</tr>
<tr>
<td>Translation Rules Written in FORTRAN</td>
<td>37</td>
</tr>
</tbody>
</table>
# TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

List Processing Primitives Used by the Compiler
# FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Representation of mode declarations by compiler lists</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>Continued</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>Run-Time Representation of Array Descriptors</td>
<td>19</td>
</tr>
</tbody>
</table>
A ONE-PASS ALGORITHM FOR COMPILING
ALGOL 68 DECLARATIONS

1. Introduction

This report describes work underway in the design of an ALGOL 68
compiler system. It presupposes some familiarity with the general ideas
of the language, such as might be gained by reading the ALGOL 68 report
(1) or some of its companion documents (2,3). In what follows, we are
primarily concerned with problems involved in compiling ALGOL 68 data
declarations, although mention is made of techniques used in other portions
of the compiler as well. Our approach is to explain the motivation behind
choosing the translation grammar in Appendix I as a means of describing
the structure and translation of programs written in the ALGOL 68 language.

The context-free grammar of Appendix I was literally modeled on the
Van Wijngaarden grammar of the report (1). That is to say, an attempt
was made to force the rules of the grammar to resemble the rules of the
report as closely as possible, except in those cases—such as the coercions
of Sections 8.2.1 through 8.2.6—where the context-free grammar was clearly
not adequate (and is not used) for descriptive purposes. In the case of
the rules of Sections 8.2.1 through 8.2.6, the actions represented by rules
in the report are to be carried out in the translation rules of the
Appendix I grammar. Our grammar diverges from the official document
principally in minor points, such as the inclusion of rules to describe
the so-called "extended language" in Section 9 of the report, and of
certain minor syntactic restrictions arising from the one-pass nature of
our algorithm, as noted below.
The translation-grammar approach used in this report is based on earlier work by Wirth and Weber (4), Lewis and Stearns (5), Schneider (6, 7), and Vere (8). The notation used in this approach can be explained in terms of the following example:

In the translation grammar of Appendix 1, the rule defining a mode declaration would be written as follows, if we could ignore the table entries and compiled code required for such a declaration:

\[
<\text{mode declaration}> +
\]

\[
\text{M\$DE}<\text{mode indicant}> = <\text{actual mode declarer}>
\]

Rule (1) is exactly the sort of rule one would expect to find in a document such as the ALOL 60 report (1) which contains a context-free grammar written in B.N.F. In order to describe the process of compiling translated code for a mode declaration, we add information to the context-free rule, as in Appendix 1:

\[
F_{25a}: <\text{mode declaration}> +
\]

\[
\text{M\$DE}/\psi_{25a1}/<\text{mode indicant}> = /\psi_{25a2}/
\]

\[
<\text{actual mode declarer}>/\psi_{25a3}/
\]

In this expansion of rule (1) above, we have added the notation "\(F_{25a}\)" to indicate that this rule is in the 25th group of rules in our grammar. We have also added three compile-time subroutines, namely, \(\psi_{25a1}, \psi_{25a2},\) and \(\psi_{25a3}\). These compile-time subroutines make entries on compile-time tables and generate code to be written out for the program scanned by the compiler.

Thus, when the symbol "M\$DE" is recognized by the compiler, subroutine \(\psi_{25a1}\) is called; when the sequence of symbols

"M\$DE <mode indicant> ="

is recognized by the compiler, subroutine \(\psi_{25a2}\) is called; and
the completion of the right-hand side of syntactic rule
\( P_{25a} \) causes subroutine \( \psi_{25a3} \) to be called.
Thus, the ensemble of rules in a translation grammar may be thought of as
an abstract representation of the program input to a compiler-compiler
system.

In fact, our translation grammar is intended to be transferred to
cards and read in to a compiler-compiler such as the one described in (6).
As can be seen in Appendix 2, the subroutines for the data declaration
section of our translation grammar have been programmed in FORTRAN V
(no less), and it is the operation and interaction of these compiler
subroutines that is the main burden of this report.
II. Techniques Used in Translation

Representation of ALGOL 68 Structures and Data Types in the Compiler

ALGOL 68 structures and invented data types are represented as linked lists in the computer memory used by the compiler. Such a representation is, naturally, machine dependent, so we propose here to give examples of how these lists are represented as storage structures on the Purdue CDC 6500 computer. Since the C.D.C. 6000 series of computers has 60-bit memory words, we chose to divide each list word into three address-size fields and one character-size field for storing miscellaneous information. An example follows, showing some typical ALGOL 68 mode declarations and their compile-time representations.

Some typical mode declarations:

MODE $S$ = PROC(INT,$S$)REAL;
MODE $SS$ = STRUCT(REF $S$,$S$),INTER, $Q$ VALUE);
MODE $SS$ = UNION($S$,$S$,$S$);
MODE $RRS$ = REF $SS$;
MODE $TT$ = [1:5, 1:6]REF $TT$;

Routines Used in Manipulating Compiler List Structures

The storage structures used in Figure 1 are similar to ones already suggested by Goos (9). To construct them, the compiler subroutines use a package of FORTRAN subroutines derived from the primitives of Weizenbaum's SLIP system (10). These SLIP-inspired primitives are listed and explained briefly in Table 1 below.
Figure 1. Representation of mode declarations by compiler lists
Figure 1. Continued
Table 1. List Processing Primitives Used by the Compiler

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LÅCF(X)</td>
<td>address (X) + LÅCF (The function returns the machine address of FORTRAN variable X.)</td>
</tr>
<tr>
<td>ÇÖNT(I)</td>
<td>[I] + ÇÖNT</td>
</tr>
<tr>
<td>INHALT(I)</td>
<td>[I] + INHALT (The value stored in the memory cell whose address is given by I is returned.)</td>
</tr>
<tr>
<td>İBYTE(J,K)</td>
<td>J(6*(K-1) + 6*K) + İBYTE (The Kth character in word J is returned right justified with padded zeroes.)</td>
</tr>
<tr>
<td>LINK(J,K)</td>
<td>J(18*(K-1) + 18*K) + LINK (The Kth address-sized field in word J is returned right justified with padded zeroes.)</td>
</tr>
<tr>
<td>PUTDIR(A,B,K)</td>
<td>A(0 + 5) + B(6*(K-1) + 6*K) (The first character of A is put into the Kth character position of word B.)</td>
</tr>
<tr>
<td>PUTIND(A,J,K)</td>
<td>A(0 + 5) + [J](6*(K-1) + 6*K) (The first character of A is put into the Kth character position of the word whose machine address is stored in J.)</td>
</tr>
<tr>
<td>SETDIR(A,B,K)</td>
<td>A(0 + 17) + B(18*(K-1) + 18*K)</td>
</tr>
<tr>
<td>SETIND(A,J,K)</td>
<td>A(0 + 17) + [J](18*(K-1) + 18*K)</td>
</tr>
</tbody>
</table>

In addition to the functions in Table 1, there is an integer function called İFETCH(K) and a logical function called SAMEFOTP(I,J,K).

The purpose of İFETCH is to return the machine address of the first word in a list cell of K words stored sequentially in computer memory. The SAMEFOTP function is, as the name suggests, a function for comparing two compiler data structures to discover if they are of the same "type". Because of the design of ALGOL 68, two declarations may be of the same
"type" if one has any number of intermixed procedure and reference prefixes followed by a copy of the second declaration. Thus, for example, if parameter I pointed to a declaration such as "PRSC REF REF PRSC REAL", and J pointed to "PRSC REAL", the SAMETYPE function would return the FORTRAN .TRUE. value. In addition, after execution of the function, the K parameter would point to a newly-created list containing the sequence "PRSC REF REF". Thus, the list returned to K indicates how many levels of dereferencing and deproceduring must be applied to a datum of type I in order to yield a datum of type J.

The SAMETYPE function also incorporates algorithms for looking up the lists assigned to mode indicators appearing in I or J and for replacing these mode indicators in either structure by pointers to these lists. No attempt is made to find reduced versions of structure declarations.

However, when two structures are compared that are structurally equivalent as defined in the ALGOL 68 report, a link to the smaller of the two structures replaces the larger structure in the compiler mode table. This structure comparison and replacement algorithm is essentially the same as the one given by Koster (11) and discussed in Goos (9).

**Intermediate Language Generated by the Compiler**

Since this report is the beginning of an attempt at producing a formal specification of ALGOL 68 that is an alternative to the Van Wijngaarden and the Vienna notations (12), it was decided that the code produced by our compiler subroutines would be in a systems language rather than in the C.D.C. COMPASS assembly language. The systems language chosen is the PILOT language of M.H. Halstead, versions of which have been implemented on the UNIVAC 1108, the C.D.C. 6000 series of computers, the IBM 360/44, and miscellaneous other machines. PILOT is a
much-distilled subset of the original NELIAC language (13), with some useful features, such as the intermixing of PILOT and machine code where desirable in a program, the use of machine addresses for indirect referencing of variable names, and the addition of partial-word masking operators. In PILOT, commas are used to separate statements, expressions can consist of at most two operands separated by an operator, and assignment of value is to the right, rather than to the left, as in ALGOL 60 or FORTRAN. A typical sequence of PILOT statements might be the following:

0 + INDEX REGISTER 1,
U - 1 + U,

LABEL 1: INDEX REGISTER 1 + 1 + INDEX REGISTER 1,
INDEX REGISTER 1 > SIZE : LABEL 2 ; ;
LACURR + 1 + [U + INDEX REGISTER 1],
LABEL 1 ,

LABEL 2:

In the statements above, index register 1 is first initialized to zero, and then a loop is entered at label 1. In the loop, index register 1 is compared to the variable "SIZE". If greater than "SIZE", the statement "LABEL 2." (go to LABEL 2) is executed. Otherwise, execution continues with the value of "LACURR + 1" stored into the memory location whose address is "U + INDEX REGISTER 1". Further examples of PILOT code are given in Section III.

Preprocessing Implied by Rules of the Grammar

When a context-free grammar is actually used as the input program of a compiler-writing system, it has to be written with greater care than one ordinarily suspects. In the first place, what is to be done with a grammar
having reserved words that resemble variable names? In Appendix 1, we see that reserved words like "REAL", "PROC", "CASE", etc. could just as well be interpreted as program variables. Next, when we look more closely, we see that the syntax for <name> involves individual letters and digits, whereas reserved words appear as groups of letters. Necessarily, the compiler generated from the grammar above will expect to scan successive words of memory, some of which contain only single letters or digits and some of which contain entire reserved words. Hence, a preprocessor subroutine is needed to read in programs to the compiler.

Along with these trivial operations of packing reserved program words for the compiler, the Appendix 1 syntax calls for the preprocessor to supply a priority digit for each expression operator encountered in a program. Since our grammar assigns unique denotations to operators, the preprocessor need only maintain a table of declared operator priorities in each scanned program block. From this table, a priority digit is inserted following each operator in programs read into the compiler. Naturally, priority declarations are no longer needed by the compiler, and so the preprocessor does not supply them to the compiler.

Stack Mechanisms Used by Compiler Subroutines

As in other compilers for block-structure programming languages, our compiler subroutines use tables to store information about currently valid program variables, labels, data types, and operators. In addition, there are two stacks whose purpose is to facilitate our single-pass translation scheme. The first stack consists of two one-dimensional FORTH arrays using the same index, "I". These arrays are referred to as "N(I)" and "TYPE(I)", respectively. As can be seen in the compiler subroutines,
the N-stack is used for constructing the compiler representations of data
structures, and the TYPE stack carries auxiliary information concerning
whether a given declaration is virtual, actual, or formal, and whether it
is stowed or nonstowed. Thus, the combined N-TYPE stack carries context
information concerning the syntactic objects recognized by the compiler.

"C\text{\textcopyright}DEL\text{\textcopyright}C" is the system name for the second stack, a one-dimensional
array with index K. As its name implies, the C\text{\textcopyright}DEL\text{\textcopyright}C stack saves informa-
tion about where nested portions of a program appear in its translation.
This information is used selectively by the compiler subroutines for
inserting and deleting sections of compiled code in the translated program
after the compiler has scanned the corresponding segment of the input
program. As will be seen in Section III, it is this use of C\text{\textcopyright}DEL\text{\textcopyright}C which
enables us to produce reasonably efficient and non-redundant code in a
single-pass compilation process. Of course, C\text{\textcopyright}DEL\text{\textcopyright}C is also used to
supply "target locations" for the jumps implicit in the translation of
conditional statements.

Using the N-stack alone, it is easy to demonstrate that a translation
grammar specifies a compiler that accepts context-sensitive languages
like \( \{ a^{n}b^{n}c^{n} : n > 0 \} \). Next, with the addition of the C\text{\textcopyright}DEL\text{\textcopyright}C stack, we
see that the resulting compiler system potentially has the same computa-
tional ability as a Turing machine. This is because any one of the com-
piler subroutines could be written so as not to return control to the
compiler, and could continue operation by manipulating the two stacks
though they were the tape of a Turing machine. Since the compiler system
can be made as general as a Turing machine, any multi-pass compiling al-
gorithm can in theory be written in a single-pass version for a transla-
tion grammar. Our goal is to write an efficient and rapid single-pass
version of what is usually considered a multi-pass compiler system.
Criteria for Selecting Rule Forms in a Translation Grammar

The criteria for selecting certain rule forms in Appendix 1 rather than others have principally to do with the one-pass nature of the compiler-writing system chosen. For example, a one-pass algorithm cannot efficiently tolerate temporary ambiguities in any subtree of a program. As a consequence, the following rules were chosen that are restrictions of the ALGOL 68 language:

\[
\begin{align*}
&\text{<mode indicant>} + $\text{name}$;
&\text{<operator>} + + | - | ... | + \text{<name>} +
&\text{<call>} + \text{<primary>} (\text{<actual parameters>})
&\text{<slice>} \rightarrow \text{<primary>} [\text{<indexers>}]
&\text{<base>} + 0 \rightarrow \text{<label>}
\end{align*}
\]

In the first two rules above, we have forced mode indicants and invented operators to be different in appearance from each other. The next two rules make it impossible to confuse a subscripted variable with a procedure call, and the last rule assures us that the statement

\[v : = \text{if a then b else c fi};\]

does not mean the same thing as

\[v : = \text{if a then b else go to c fi};\]

which is currently a valid interpretation in the ALGOL 68 report.

Again, the desire for an uncomplicated single-scan compiler led us to abandon the ALGOL 68 definition of a block (given in Section 6.1.1 of the report) in favor of a restricted subset of the definition. In the full ALGOL 68 version of a block, the declaration prelude terminates at the first \text{\texttt{labeled}} statement or the first statement followed by a jump. To translate such a declaration prelude properly, the compiler needs to know at the beginning of each statement in a declaration prelude what
comes after that statement. Such knowledge could be gained by a separate
pass or by requiring forward scans before each statement, but this extra
work consumes compiler time that could be better spent doing translation.
Hence, our version of the ALGOL 68 block (rules P34 and P35 in Appendix 1)
is a subset of the full version in which the declaration prelude terminates
at the first executable statement in the block. No generality is lost in
this version by not interspersing statements and declarations in the
declarations in the blocks that legally may appear there.

The preceding problems of declaration preludes and uniquely identifi-
able mode and operator indicants are all examples of avoiding syntaxes
that result in nondeterministic compilers (6). Another example of this
problem is one that arose when an attempt was made to transcribe a portion
of the ALGOL 68 report directly into the notation of Appendix 1. In this
case the syntax for <stowed declarer> initially took the form:

<stowed declarer> = STRUCT (<fields>)
   | [<rows>] STRUCT (<fields>)
   | [<rows>] <nonstowed declarer>
   | [<rows>] <mode indicant>

With this set of rules, we are left with three separate methods to declare
an array and no way of knowing initially which compiler subroutines
should be called. Because of this, the following version of the syntax
of arrays and structures appears in Appendix 1:
<stowed declarer> + STRUCT (<fields>)
  | <array generator> STRUCT (<fields>)
  | <array generator> <nonstowed declarer>
  | <array generator> <mode indicant>
<array generator> + [<rows>]

Apart from the necessity of avoiding nondeterminisms in the compiler, the other main criterion for selecting certain configurations of rules in a translation grammar is to permit syntax-directed modifications of compiler tables and syntax-directed tests for consistency of program subtrees with non-syntactic objects. As an example of simple modifications made possible by syntax, the following translation rules cause an extra level of reference to be prefixed on the compile-time representation of the declarer in question:

\[ F_{29\alpha} : <\text{reference to actual declarer}> + <\text{actual mode declarer}> /\psi_{29\alpha}/ \]

\[ F_{2\alpha} : <\text{nonstowed declarer}> + \text{REF}/\psi_{2\alpha}/<\text{mode declarer}> /\psi_{2\alpha}/ \]

\[ F_{30\alpha} : <\text{reference to mode global generator}> + \text{HEAP}/\psi_{30\alpha}/ <\text{actual mode declarer}> /\psi_{30\alpha}/ \]

In the second example, the rules

\[ F_{21\alpha} : <\text{virtual mode declarer}> + <\text{mode declarer}> /\psi_{21\alpha}/ \]

\[ F_{22\alpha} : <\text{actual mode declarer}> + <\text{mode declarer}> /\psi_{22\alpha}/ \]

do not introduce any syntactic ambiguity in the language, since the corresponding transitions of the compiler can only occur in mutually exclusive contexts. However, compiler subroutines \( \psi_{21\alpha} \) and \( \psi_{22\alpha} \) both test the N-TYPE stack to see that all arrays declared within the \(<\text{mode declarer}>\) are respectively virtual or actual.
Lastly, we see that syntactic rules may be chosen to make the work of compiler subroutines easier. Good examples of this are the sets of rules below:

\[ P_{5a}: \text{<fields>} + \text{<field>}/\psi_{5a} \]
\[ P_{5b}: \text{<fields>} + \text{<field}, <fields>}/\psi_{5b} \]
\[ P_{6a}: \text{<field>, <field>}/\psi_{6a} \]

Subroutine \( \psi_{6a} \) generates a compile-time list cell for each field in a structure declaration, and subroutine \( \psi_{5b} \) links together the list cells generated in this fashion. Because of the form of \( P_{6a} \), the structure fields are stored in sequence on the N-TYPE stack by \( \psi_{6a} \). Rule \( P_{5b} \) calls for the compiler to work backwards, linking together the two structure fields on top of the N-TYPE stack by successive calls on subroutine \( \psi_{5b} \). Thus the syntax directs the compiler to link together the compile-time representation of the structure.

On the other hand, the rules

\[ P_{7a}: \text{<rows> + <row>}/\psi_{7a} \]
\[ P_{7b}: \text{<rows> + <rows>, <row>}/\psi_{7b} \]

do nothing more than construct a run-time array descriptor in the order of appearance of the program.
III. Translation of Array and Structure Declarations

Preliminaries

ALGOL 68 uses two reservoirs of consecutive memory words for use in generating arrays and structures during program execution. In the first reservoir system, array storage is taken from a stack (called loc in the defining report) whose contents are divided into regions corresponding to currently active program blocks. As the most recent block is abandoned by the program at run time, the storage of its corresponding region in loc is freed for reuse, exactly as in an ALGOL 60 system. As a consequence, structures and arrays stored in loc are only accessible during the lifetime of the block in which they were created. In order to permit a longer lifetime of usage for selected arrays and structures, an alternative reservoir system is available, called the heap in the defining report. The heap system provides that an array or structure is only reclaimed when it can no longer be reached from an active program variable. The use of this heap reservoir necessitates use of a conventional list-processing storage reclamation scheme at run time.

During compilation, the compiler subroutines keep track of whether a currently translated declaration is to be stored in loc or heap. The bookkeeping system used for this purpose is quite simple. The FORTRAN variable "LHCURR" is initialized to contain the characters "L6C", and is temporarily reset to contain the characters "HEAP" whenever compiler subroutines $\Psi_{30\alpha1}$ and $\Psi_{30\alpha2}$ in translation rule $P_{30\alpha}$ are called. Then, during the translation process, whenever the code for a storage reservation is put out, the contents of LHCURR (denoted in our notation below by "[LHCURR]")) is inserted in the translated program.
Array Declarations

There are two parts to the problem of generating arrays in a compiled ALGOL 68 program. The first part involves construction of a run-time descriptor for the array, and the second part involves possible assignment of structures to each element of the resulting array. Translation rules $P_{4a}$ and $P_{7a}$ of Appendix 1 outline the construction of the descriptor, while rules $P_{10a}$ through $P_{13e}$ keep track of whether the descriptor is virtual (has no numerical bounds), actual (has only numerical bounds) or formal (has some mixture of numerical and non-numerical bounds). Since no code is produced for virtual or formal descriptors, compiler subroutine $\psi_{4a1}$ records in the GDELOC stack where translated code for the descriptor begins, so that $\psi_{4a2}$ can erase any code that may have been produced while scanning the <array generator>.

For the recognized program sequence

[<rowers>]

the compiler produces the following translated code:

SAVEJ,
2 + (LACURR) → U,
(LACURR) + \[1 + \text{no. of dimensions}\] → (LACURR),
O → [U],
Translation of <rowers>.
U = \[1 + \text{no. of dimensions}\] → TEMP1,
O → [TEMP1],
[U](36 + 53) → SIZE,
SIZE > 0: U + [TEMP1](0 + 17),
(LACURR) + SIZE + (LACURR);;
\[\text{no. of dimensions}\] → [TEMP1](36 + 53),
P\&FU,
The run-time variable \( U \) in the above code is used by the compiler sub-
routines as an index for placing information in the current word of the
array descriptor being generated. As can be seen in Figure 2, the de-
scriptor consists of \( n+1 \) words, with \( n \) the number of dimensions. Thus,
the first descriptor word storing bounds is located at address
"2 + \{LHCURR\}" above (see Preliminaries for meaning of \{LHCURR\}), and
that word is set to zero by the "0 + \{U\}" statement.

The statement that reserves space for the descriptor in \{LHCURR\} is
inserted later by subroutine \( \psi_{H2} \) after the number of dimensions is known
by the compiler. Translation of \<rowers\> involves calculation of the
"strides" of the descriptor. Each stride is associated with a pair of
bounds and is the coefficient in a run-time storage-mapping function used
in subscripting. When the final stride of the descriptor is calculated,
it is stored in the next location following the last pair of bounds in
memory and coincides with the actual size of the run-time array. Thus,
the code

\[
[U](36 + 53) + \text{SIZE},
\]
above extracts this value and uses it above in a conditional statement
that places a link to the first array element into the run-time descriptor:

\[
\text{SIZE} > 0: \text{U} + [\text{TEMP}](0 + 17),\{\text{LHCURR}\} + \text{SIZE} + \{\text{LHCURR}\}.
\]

Figure 2 below displays the storage structure used for run-time array
descriptors. The "\( i_4, i_1 \)" notation represents a two-bit code for indicating
whether or not the corresponding lower and upper bounds can be changed
dynamically. When \( i_4 \) (or \( i_1 \)) is zero, the lower (or upper) bound may vary;
otherwise it is fixed. Given this information, we can discuss the trans-
lation of \<rowers\>. 
Figure 2. Run-Time Representation of Array Descriptors

<table>
<thead>
<tr>
<th>Type</th>
<th>No. of Dimensions</th>
<th>Block Number</th>
<th>Pointer to First Element of Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>(_l_1)</td>
<td>stride 1</td>
<td>upper bound 1</td>
<td>lower bound 1</td>
</tr>
<tr>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td>(\ldots)</td>
</tr>
<tr>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td>(\ldots)</td>
</tr>
<tr>
<td>(_l_n)</td>
<td>stride n</td>
<td>upper bound n</td>
<td>lower bound n</td>
</tr>
</tbody>
</table>

Rules \(P_{7a}\) and \(P_{7b}\) of Appendix 1 are given below:

\[ P_{7a} : \text{<rowers>} + <\text{rower}>/\psi_{7a} \]

\[ P_{7b} : \text{<rowers>} + <\text{rowers}>, <\text{rower}>/\psi_{7b} \]

Subroutine \(\psi_{7a}\) is called after the translation of the first pair of bounds in an array declaration, and subroutine \(\psi_{7b}\) is called for each subsequent pair of bounds in the declaration. Primarily, \(\psi_{7a}\) and \(\psi_{7b}\) work together to count dimensions and calculate the strides of the descriptor. They assume that translation rules \(P_{13}, P_{12}, P_{11},\) and \(P_{10}\) have inserted the upper and lower bounds of each dimension into the appropriate fields of the descriptor word. The code put out by \(\psi_{7b}\) gives an idea of this interaction:
\[ U(18 + 35) - [U(0 + 17) + \text{SIZE}, \]

(Subtracts the lower bound from the upper bound.)

\[ \text{SIZE} \times [U(36 + 53) + \text{SIZE}, \]

(Calculates the value of the next stride.)

\[ U + 1, U, \]

\[ 0 \rightarrow [U], \]

(Zeroes the next word in the descriptor.)

\[ \text{SIZE} > 0 : \text{SIZE} \rightarrow [U(36 + 53)]; \]

(Stores the stride in the next word of the descriptor.)

When the array descriptor is completed, there may follow code for initializing each element of the resulting run-time array. This code is written out for rules \( P_{3b} \) and \( P_{3d} \):

\[ P_{3b} : \text{<stowed declarer> + <array generator>/$\psi_{3b1}$/} \]

\[ \text{STRUCT}(/\psi_{3b2}/ \text{<fields>})/\psi_{3b3} \]

\[ P_{3d} : \text{<stowed declarer> + <array generator>/$\psi_{3d1}$/} \]

\[ \text{<mode indicant>/$\psi_{3d2}$/} \]

Essentially, subroutines \( \psi_{3d1} \) and \( \psi_{3d2} \) set up a loop that links each element of the generated array to a fresh copy of the \text{<mode indicant>}. This copy of the \text{<mode indicant>} is created by treating the \text{<mode indicant>} as a function whose value is the address of the next location of the storage reservoir specified by LHCURR. Since translation of \( P_{3b} \) works in a very similar fashion, we will give as our illustration the code produced by \( \psi_{3d1} \) and \( \psi_{3d2} \).
0 \rightarrow \text{INDEX REGISTER} 2,
U - 1 + U,

\text{LABEL 2}:
\text{INDEX REGISTER} 2 + 1 \rightarrow \text{INDEX REGISTER} 2,
\text{INDEX REGISTER} 2 > \text{SIZE}: \text{LABEL 3};
[LH\text{CURR}] + 1 + [U + \text{INDEX REGISTER} 2],
\text{Translation of }<\text{mode indicant}>,
(This includes a jump to the definition of the
<mode indicant> and return.)
\text{LABEL 2}.,

\text{LABEL 3}:
Note in the above code that "U - 1" denotes the first address in the
region of memory reserved for the stored array, whereas \(LH\text{CURR}\) was set
equal to the last address of this region by compiler subroutine \(\Psi_{ba2}\).
Thus, the first address in any data structure created by the <mode
indicant> is given by "([LH\text{CURR}] + 1", and this address is the value used
to initialize the corresponding array elements iteratively.

\text{Structure Declarations}

In our implementation of ALGOL 68, the fields of data structure can
store plain values or they can be linked to other structures and arrays.
During a declaration, then, code linking a structure field to another
structure or array may or may not be called for. On a syntactic basis,
we say that such linking code is called for in a structure having some
field that corresponds to a <stowed declarer> or <mode indicant>. The
code is not compiled in the case of a structure field that corresponds to
a <nonstowed declarer>. To simplify our discussion of the translation
process, we consider only rule 1' of Appendix 1, since it uncs
essentially the same mechanisms as rule \( P_{3b} \):

\[ P_{3b} : \langle \text{stowed declarer} \rangle \rightarrow \text{STRUCT(} \langle \psi_{3a1} \langle \text{fields} \rangle \rangle \langle \psi_{3a2} \rangle \rangle \]

The code generated by compiler subroutines \( \psi_{3a1} \) and \( \psi_{3a2} \) is as follows:

\[
\text{SAVE } U, \\
(LHCURR) \rightarrow U, \\
(LHCURR) + \text{no. of fields} \rightarrow \{LHCURR\}, \\
\text{Translation of } \langle \text{fields} \rangle, \\
P\&P \ U, \\
\]

The compiler subroutines used above in translating \( \langle \text{fields} \rangle \) also count the number of fields in the structure. When the count is completed, subroutine \( \psi_{3a2} \) is called, and \( \psi_{3a2} \) inserts the field count into the instruction preceding the translation of \( \langle \text{fields} \rangle \). Thus, the code produced by \( \psi_{3a1} \) and \( \psi_{3a2} \) reserves space for the translated structure in L\&C or HEAP and initializes the index of structure fields to point to the first field in the structure to be translated. The code put out for a typical "stowed" field of the structure is produced by compiler subroutines \( \psi_{3a1} \) or \( \psi_{6a} \) as follows:

\[
\text{no. of intervening nonstowed fields} + U \rightarrow U, \\
(LHCURR) + 1 \rightarrow \{U\}, \\
\text{Translation of } \langle \text{field} \rangle, \\
\]

In the code above, the index \( U \) retains the address of the most recently initialized field in the compiled structure. In order to initialize the next field, \( U \) is incremented, and the address of the first free word in L\&C or HEAP is inserted in the word whose address is \( U \). Thus, the run-time field is made to point to the first address in the generated structure corresponding to the translated \( \langle \text{field} \rangle \). Of course, no code at all will appear for a "nonstowed" field.
IV. Translation of Mode Declarations

As seen in rule $P_{25a}$, a mode declaration has the form

\[
\text{MODE <mode indicant> = <actual mode declarer>}. \]

Sample code produced for such a declaration by compiler subroutines $\psi_{25a1}$, $\psi_{25a2}$, and $\psi_{25a3}$ is as follows:

\[
\begin{align*}
\text{LABEL 5:} & \\
\text{LABEL 4:} & \\
\text{Translation of <actual mode declarer>,} & \\
\text{\textbf{RETURN}} & , \\
\text{LABEL 5:} & \\
\end{align*}
\]

In the above code, "LABEL 4" is the unique program label corresponding to the definition of the <mode indicant>. "LABEL 5" is used to isolate the indicant definition in the run-time program. With this isolation, the only way to execute the code produced for <actual mode declarer> is to jump to LABEL 4. After this declarer is executed at run-time, there follows the instruction

\[
\text{\textbf{RETURN}},
\]

which is used in the PILOT language for a jump to the location whose address is stored in RETURN.

In order to execute this data-definitional function, we need a function call in the form of a <mode indicant> appearing in some declaration. This function call produces the following code:

\[
\begin{align*}
\text{SAVE RETURN,} & \\
J + 2 \rightarrow \text{RETURN,} & \\
\text{LABEL 4:} & \\
\text{POP RETURN} & .
\end{align*}
\]
For this calling sequence, LABEL 4 is used as the location of the particular <mode indicant> to be called. The PILOT run-time program counter stored in J is saved in the RETURN variable so that the code for the mode definition can jump back to the statement "P$P RETURN". All the compiler needs to do is carry a record of the labels that correspond to the currently active mode indicants and to retain a representation of the data structures to which these indicants correspond. As before, the value of this mode indicant function is the first free location in L$C or H$AP at the calling point in the compiled program.

A Final Example

To bring together the information given in our previous descriptions, we present a typical mode declaration, together with its generated code:

The Declaration

MODE $A$ = STRUCT([1:5] STRUCT($B$ X, REAL Y));

The Compiled Code

LABEL 10.,
LABEL 7:
SAVE U,
L$C + U,
L$C + 1 + L$C,
U + 1 + U,
L$C + [U],
SAVE U,
2 + L$C + U,
L$C + 2 + L$C,
0 + [U],
1 + [U](36 + 53),
1 + OPERAND REGISTER 2,
1 + [U](55 + 55),
OPERAND REGISTER 2 + [U](0 + 17),
5 + OPERAND REGISTER 2,
1 + [U](54 + 55),
OPERAND REGISTER 2 + [U](18 + 35),
1 + [U](36 + 53),
[U](14 + 35) - [U](0 + 17) \rightarrow SIZE,
SIZE + 1 \rightarrow SIZE,
U + 1 + U,
0 + [U],
SIZE > 0: SIZE + [U](36 + 53);;
U - 2 \rightarrow \text{TEMP1},
0 - [\text{TEMP1}],
[u](36 + 53) + \text{SIZE},
\text{SIZE} > 0: U + [\text{TEMP1}](0 + 17),
L\text{OC} + \text{SIZE} \rightarrow L\text{OC};;
1 + [\text{TEMP1}](36 + 53),
P\text{POP} U,
0 + \text{INDEX REGISTER \#3},
U - 1 \rightarrow U,
\text{LABEL 8}:
\text{INDEX REGISTER \#3} + 1 + \text{INDEX REGISTER \#3},
\text{INDEX REGISTER \#3} > \text{SIZE}; \text{LABEL 9};;
L\text{OC} + 1 \rightarrow [U + \text{INDEX REGISTER \#3}],
\text{SAVE} U,
L\text{OC} + U,
L\text{OC} + 2 \rightarrow L\text{OC},
1 + U \rightarrow U,
L\text{OC} + 1 \rightarrow [U],
\text{SAVE RETURN},
J + 2 \rightarrow \text{RETURN},
\text{LABEL ($\#$)};,
P\text{POP} \text{RETURN}
P\text{POP} U,
\text{LABEL 8}.,
\text{LABEL 9}:
P\text{POP} U,
[\text{RETURN}] .
\text{LABEL 10}:
References Cited in Text


   (July 1968), 9-49.

3. Peck, J. E. L. An ALGOL 68 companion. Draft paper, University of

   formal definition: parts I and II. Comm. ACM 3 (Jan. - Feb.

5. Lewis, P. M. and Stearns, R. E. Syntax-directed transduction.
   J. ACM 12 (July 1965), 465-488.

6. Schneider, V. B. A system for designing fast programming language

7. Schneider, V. B. A translation grammar for ALGOL 68. Proc. Spring


9. Goos, G. Some problems in compiling ALGOL 68. Technical Report,
   Rechenzentrum der Technischen Hochschule, Muenchen, Germany,
   1970.

    524.

    86-89.

12. Lucas, P. and Walk, K. On the formal description of PL/1. In Annual

Appendix 1

A Partial Translation Grammar for ALGOL 68*

\[ P_{1a} : \text{\langle mode declarer\rangle } \rightarrow \text{\langle stowed declarer\rangle }/P_{1a}\]
\[ P_{1b} : \text{\langle nonstowed declarer\rangle }/P_{1b}\]
\[ P_{1c} : \text{\langle mode indicator\rangle }/P_{1c}\]
\[ P_{2a} : \text{\langle nonstowed declarer\rangle } \rightarrow \text{\langle REF\rangle }/P_{2a}\]
\[ P_{2b} : \text{\langle mode declarer\rangle }/P_{2b}\]
\[ P_{3a} : \text{\langle stowed declarer\rangle } \rightarrow \text{\langle STRUCT\rangle }(/P_{3a}\]
\[ P_{3b} : \text{\langle array generator\rangle }/P_{3b}\]
\[ P_{3c} : \text{\langle array generator\rangle } \rightarrow \text{\langle nonstowed declarer\rangle }/P_{3c}\]
\[ P_{3d} : \text{\langle array generator\rangle } \rightarrow \text{\langle mode indicator\rangle }/P_{3d}\]
\[ P_{4a} : \text{\langle array generator\rangle } \rightarrow [/P_{4a} \text{\langle rows\rangle }]/P_{4a}\]
\[ P_{5a} : \text{\langle fields\rangle } \rightarrow \text{\langle field\rangle }/P_{5a}\]
\[ P_{5b} : \text{\langle field\rangle } \rightarrow \text{\langle field\rangle }/P_{5b}\]
\[ P_{6a} : \text{\langle field\rangle } \rightarrow \text{\langle field\rangle } \rightarrow \text{\langle mode declarer\rangle }/P_{6a}\]
\[ P_{7a} : \text{\langle rows\rangle } \rightarrow \text{\langle row\rangle }/P_{7a}\]
\[ P_{7b} : \text{\langle row\rangle } \rightarrow \text{\langle row\rangle }/P_{7b}\]
\[ P_{8a} : \text{\langle field mode declarer\rangle } \rightarrow \text{\langle mode declarer\rangle }/P_{8a}\]
\[ P_{9a} : \text{\langle field selector\rangle } \rightarrow \text{\langle name\rangle }/P_{9a}\]
\[ P_{10a} : \text{\langle row\rangle } \rightarrow :/P_{10a}\]
\[ P_{10b} : \text{\langle lower bound\rangle } \rightarrow \text{\langle upper bound\rangle }/P_{10b}\]
\[ P_{11a} : \text{\langle lower bound\rangle } \rightarrow \text{\langle bound\rangle }/P_{11a}\]
\[ P_{12a} : \text{\langle upper bound\rangle } \rightarrow \text{\langle bound\rangle }/P_{12a}\]
\[ P_{13a} : \text{\langle bound\rangle } \rightarrow \text{\langle formula\rangle }/P_{13a}\]
\[ P_{13b} : \text{\langle formula\rangle } \rightarrow \text{\langle either\rangle }/P_{13b}\]
\[ P_{13c} : \text{\langle either\rangle } \rightarrow \text{\langle FLEX\rangle }/P_{13c}\]

* This syntax has been tested, and is certified to be LR(1).
$P_{13}^a$: `<formula EITHER/$P_{13}d$/`

$P_{13}e^a$: `<union mode decl>`

$P_{14}a^b$: `<mode decarator> + <procdecl>`

$P_{14}b^c$: `<virtual procplan>`

$P_{14}c^d$: `<virtual parameters>`

$P_{14}d^e$: `<mode parameter>`

$P_{14}e^f$: `<mode parameter>`

$P_{14}f^g$: `<mode parameter>`

$P_{15}a^h$: `<procdecl> + PROC/$P_{15}a$/`

$P_{15}b^i$: `<virtual procplan>`

$P_{16}a^j$: `<virtual parameters>`

$P_{16}b^k$: `<virtual mode declarer>`

$P_{16}c^l$: `<virtual parameters>`

$P_{17}a^m$: `<virtual parameters>`

$P_{17}b^n$: `<virtual mode declarer>`

$P_{18}a^o$: `<union mode decl>`

$P_{19}a^p$: `<mode indicant>`

$P_{20}a^q$: `<mode indicant>`

$P_{21}a^r$: `<mode declarer>`

$P_{22}a^s$: `<actual mode declarer>`

$P_{23}a^t$: `<formal mode declarer>`

$P_{24}a^u$: `<declaration>`

$P_{24}b^v$: `<operator declaration>`

$P_{24}c^w$: `<initialization>`

$P_{24}d^x$: `<priority declaration>`
\[ \text{P}_{25a} : \text{<mode declaration>} \rightarrow \text{MODE/}^{{\text{25a1}}} / \text{<mode indicant>} \rightarrow /^{{\text{25a2}}} / \\
\text{<actual mode declamer/>} /^{{\text{25a3}}} / \\
\text{P}_{26a} : \text{<operator declaration>} \rightarrow \text{OP/}^{{\text{26a1}}} / \text{<op.>} \rightarrow /^{{\text{25a2}}} / \text{/routine/>} /^{{\text{26a3}}} / \\
\text{P}_{27a} : \text{<initialization>} \rightarrow \text{<reference to actual declamer/>} /^{{\text{27a}}} / \text{/<name list>} \\
\text{P}_{27b} : \text{<reference to mode global generator/>} /^{{\text{27b}}} / \text{/<name list>} \\
\text{P}_{27c} : \text{<formal mode declamer/>} \text{<tag>} \rightarrow /^{{\text{27c1}}} / \text{/<tertiary/>} /^{{\text{27c2}}} / \\
\text{P}_{28a} : \text{<name list>} \rightarrow \text{<tag/>} /^{{\text{28a}}} / \\
\text{P}_{28b} : \text{<tag>} \rightarrow /^{{\text{28b1}}} / \text{/<tertiary/>} /^{{\text{28b2}}} / \\
\text{P}_{28c} : \text{<name list>, <tag/>} /^{{\text{28c}}} / \\
\text{P}_{28d} : \text{<name list>, <tag/>} \rightarrow /^{{\text{28d1}}} / \text{/<tertiary/>} /^{{\text{28d2}}} / \\
\text{P}_{29a} : \text{<reference to actual declamer/>} \rightarrow \text{<actual mode declamer/>} /^{{\text{29a}}} / \\
\text{P}_{30a} : \text{<reference to mode global generator/>} \rightarrow \text{HEAP/}^{{\text{30a1}}} / \text{/<actual mode declamer/>} /^{{\text{30a2}}} / \\
\text{P}_{31a} : \text{<program> \rightarrow $ENTRY$/}^{{\text{31a1}}} / \text{/<particular program>$/EXIT$/}^{{\text{31a2}}} / \\
\text{P}_{32a} : \text{<particular program> \steparrow <closed clause>} \\
\text{P}_{33a} : \text{<closed clause> \steparrow BEGIN/}^{{\text{33a1}}} / \text{/<serial clause>$/END/$}^{{\text{33a2}}} / \\
\text{P}_{33b} : \text{(/}^{{\text{33a1}}} / \text{/<serial clause>} /)^{\text{33a2}} / \\
\text{P}_{33c} : \text{(/}^{{\text{33a1}}} / \text{/<row of clause>} /)^{\text{33a2}} / \\
\text{P}_{34a} : \text{<serial clause> \steparrow <declaration prelude sequence>; <suite of clause train> \\
\text{P}_{34b} : \text{<declaration prelude sequence>;<label completer><suite of clause train> \\
\text{P}_{34c} : \text{<label completer><suite of clause train> \\
\text{P}_{34d} : \text{<suite of clause train> \\
\text{P}_{35a} : \text{<declaration prelude> \steparrow <unitary clause>;<declaration prelude> \\
\text{P}_{35b} : \text{<declaration> \\
\text{P}_{36a} : \text{<declaration prelude sequence>;< <declaration prelude sequence> \\
\text{P}_{36b} : \text{<unitary clause>;< <unitary clause> \\
\text{P}_{36c} : \text{<unitary clause EXIT> <unitary clause>EXIT \\
\text{P}_{36d} : \text{<label> <label>} ;}
\[ P_{37a}: \quad \text{<suite of clause train>} \rightarrow \text{<unitary clause>}. \]

\[ P_{37b}: \quad |\text{<labelstat>}. \]

\[ P_{37c}: \quad |\text{<labelstat><suite of clause train>}. \]

\[ P_{37d}: \quad |\text{<unitary clause;><suite of clause train>}. \]

\[ P_{38a}: \quad |\text{<labelstat> + <unitary clause;><label>:>}. \]

\[ P_{38b}: \quad |\text{<unitary clause EXIT><label>:>}. \]

\[ P_{38c}: \quad |\text{<labelstat><label>:>}. \]

\[ P_{38d}: \quad |\text{<label completer -> <label>:>}. \]

\[ P_{38e}: \quad |\text{<label completer><label>:>}. \]

\[ P_{38f}: \quad <\text{declaration prelude sequence}> \rightarrow <\text{declaration prelude}>/P_{34b2}/. \]

\[ P_{38g}: \quad |\text{<declaration prelude sequence;><declaration prelude>}. \]

\[ P_{39a}: \quad <\text{label}> + <\text{name}>. \]

\[ P_{40a}: \quad <\text{unitary clause}> + <\text{tertiary}>. \]

\[ P_{40b}: \quad |\text{<confrontation>}. \]

\[ P_{40c}: \quad |\text{FOR <tag> FROM <serial clause> BY <serial clause> TO <serial clause>}
\quad \text{DØ <unitary clause>}. \]

\[ P_{40d}: \quad |\text{FROM <serial clause> BY <serial clause> TO <serial clause> DØ}
\quad <\text{unitary clause>}. \]

\[ P_{40e}: \quad |\text{WHILE <serial clause> DØ <unitary clause>}. \]

\[ P_{41a}: \quad <\text{confrontation}> \rightarrow <\text{identity relation}>. \]

\[ P_{41b}: \quad |\text{<conformity relation>}. \]

\[ P_{41c}: \quad |\text{<reference to mode assignation>}. \]

\[ P_{41d}: \quad |<\text{mode cast}>. \]

\[ P_{42a}: \quad <\text{reference to mode assignation}> \rightarrow <\text{reference to mode tertiary>}
\quad : = <\text{unitary clause>}. \]

\[ P_{43a}: \quad <\text{reference to mode tertiary}> \rightarrow <\text{tertiary}>. \]

\[ P_{44a}: \quad <\text{conformity relation}> \rightarrow <\text{reference to mode tertiary>}
\quad <\text{conformity relator><tertiary>}. \]
\[\begin{align*}
P_{45a}: & \quad \text{<conformity relator> } \rightarrow :: \\
P_{45b}: & \quad |::= \\
P_{46a}: & \quad \text{<identity relations> } \rightarrow \text{<reference to mode tertiary>
}
\quad \text{(reference to mode tertiary)} \\
P_{47a}: & \quad \text{<identity relator> } \rightarrow :: \\
P_{47b}: & \quad |::= \\
P_{48a}: & \quad \text{<mode cast> } \rightarrow \text{<virtual mode declarer>/}48a1/\text{<unitary clause>/}48a2/ \\
P_{49a}: & \quad \text{<void cast> } \rightarrow \text{<unitary clause>/}49a/ \\
P_{50a}: & \quad \text{<tertiary> } \rightarrow \text{<formula>}
\end{align*}\]

\[\begin{align*}
P_{51a}: & \quad \text{<formula> } \rightarrow \text{<formula><op.<binary>}
\quad |\text{<binary>}
\end{align*}\]

\[\begin{align*}
P_{52a}: & \quad \text{<binary> } \rightarrow \text{<op.><binary>}
\quad |\text{<secondary>}
\end{align*}\]

\[\begin{align*}
P_{53a}: & \quad \text{<priority declaration> } \rightarrow \text{PRIORITY<op>=<nonzero digit>}
\end{align*}\]

\[\begin{align*}
P_{54a}: & \quad \text{<void cast> } \rightarrow \text{<mode cast>}
\quad |\text{<void cast>}
\end{align*}\]

(Rules \(P_{55}\) through \(P_{60}\) of first version are deleted.)

\[\begin{align*}
P_{61a}: & \quad \text{<op.> } \rightarrow + \\
P_{61b}: & \quad | \rightarrow - \\
P_{61c}: & \quad | \rightarrow * \\
P_{61d}: & \quad | \rightarrow / \\
P_{61e}: & \quad | \rightarrow ./
\end{align*}\]

\[\begin{align*}
P_{61f}: & \quad | \rightarrow + \\
P_{61g}: & \quad | \rightarrow - \\
P_{61h}: & \quad | \\
P_{61i}: & \quad | \rightarrow ) \\
P_{61j}: & \quad | \rightarrow 
\end{align*}\]
\[ P_{61k} : | > \]
\[ P_{61i} : | < \]
\[ P_{61m} : | \geq \]
\[ P_{61n} : | \leq \]
\[ P_{61o} : | = \]
\[ P_{61p} : | \# \]
\[ P_{61q} : | -:= \]
\[ P_{61r} : | +:= \]
\[ P_{61s} : | *:= \]
\[ P_{61t} : | /:= \]
\[ P_{61u} : | ./.:= \]
\[ P_{61v} : | \text{RE} \]
\[ P_{61w} : | \text{IM} \]
\[ P_{61x} : | \text{ODD} \]
\[ P_{61y} : | \text{SIGN} \]
\[ P_{61z} : | \text{ROUND} \]
\[ P_{61aa} : | +\text{<name>\#} \]
\[ P_{62a} : \text{<secondary> + <primary>} \]
\[ P_{62b} : | \text{<cohesion>} \]
\[ P_{63a} : \text{<cohesion> + <selection>} \]
\[ P_{63b} : | \text{<generator>} \]
\[ P_{64a} : \text{<selection> + <name> \$F \text{<secondary>} \]
\[ P_{65a} : \text{<generator> + <reference to mode global generator>} \]
\[ P_{65b} : | \text{<reference to mode local generator>} \]
\[ P_{66a} : | \text{<reference to mode local generator> + LOC <actual mode declarer>} \]
\[ P_{67a} : \text{<primary> + <base>} \]
\[ P_{67b} : | \text{<closed clause>} \]
\[ P_{67c} : | \text{<conditional clause}> \]
\[ P_{68a} : \text{<base>} + \text{<tag>} \]
\[ P_{68b} : \text{<denotation>} \]
\[ P_{68c} : \text{<slice>} \]
\[ P_{68d} : \text{<call>} \]
\[ P_{68e} : \text{GO} \text{ TO} \text{ <label>} \]
\[ P_{68f} : \text{SKIP} \]
\[ P_{68g} : \text{NIL} \]
\[ P_{69a} : \text{<tag>} + \text{<name>} \]
\[ P_{70a} : \text{<name>} + \text{<letter>} \]
\[ P_{70b} : \text{<name>} \text{<letter>} \]
\[ P_{70c} : \text{<name}> \text{<digit>} \]
\[ P_{70d} : \text{<name}*} \]
\[ P_{71a} : \text{<denotation> + <number>} \]
\[ P_{71b} : \text{<character denotation>} \]
\[ P_{71c} : \text{<string denotation>} \]
\[ P_{71d} : \text{<bits>} \]
\[ P_{71e} : \text{<routine>} \]
\[ P_{72a} : \text{<routine>} \to \text{(*\text{<formal parameters>}\text{<moid case>*})} \]
\[ P_{72b} : \text{<moid cast pack>} \]
\[ P_{73a} : \text{<moid cast pack>} \to \text{(*\text{/73a1/<moid cast>*})/73a2/} \]
\[ P_{74a} : \text{<formal parameters>} + \text{<formal parameters>} \]
\[ P_{74b} : \text{<formal parameter> \to <formal parameter>;} \]
\[ P_{75a} : \text{<formal parameter> \to <formal parameter>;} \]
\[ P_{76a} : \text{<formal parameter> + <formal mode declarer> <name>} \]
\[ P_{77a} : \text{<call> \to <primary>\text{/77a1/<actual parameters>*}/77a2/} \]
\[ P_{78a} : \text{<actual parameters> \to <unitary clause>/78a1/} \]
\[ P_{78b} : \text{<unitary clause>/78b1/<actual parameters>*}/78b2/ \]
\[ P_{79a} : \text{<slice> \to <primary> \text{[-indexer]}}} \]
```pascal
P_80a: <indexer> + <trimsclip>  
P_80b: |<indexer>, <trimsclip>  
P_81a: <trimsclip> + <sum>  
P_81b: |: AT <sum>  
P_81c: |<sum> : <sum> AT <sum>  
P_81d: |<sum> : AT <sum>  
P_81e: |<sum> : <sum>  
P_81f: |: <sum> AT <sum>  
P_82a: <conditional clause> + IF/\textasciitilde35a/ <serial clause>  
   THEN/\textasciitilde82a2/ <serial clause>  
   ELSE/\textasciitilde82a3/ <serial clause> Fl/\textasciitilde82a4/  
P_82b: |IF/\textasciitilde35a/ <serial clause> THEN/\textasciitilde82a2/ <serial clause> Fl/\textasciitilde82b3/  
P_82c: |CASE/\textasciitilde35a/ <serial clause> IN/\textasciitilde82d2/ <row of clause> ESAC/\textasciitilde82c3/  
P_82d: |CASE/\textasciitilde35a/ <serial clause> IN/\textasciitilde82d2/ <row of clause> ELSE/\textasciitilde82d3/  
   <serial clause> ESAC/\textasciitilde82d4/  
P_83a: <row of clause> + <unitary clause>, <row of clause>/\textasciitilde83a/  
P_83b: |<unitary clause>, <row of clause>/\textasciitilde83b/  
P_84a: <unitary clause>, <row of clause>/\textasciitilde84a/  
P_85a: Deleted  
P_86a: <number> + <a integer>  
P_86b: |<a integer> , <b integer>  
P_86c: |<a integer>  
P_86d: |<b integer>  
P_87a: <a integer> + <nonzero digit>  
P_87b: |<a integer> <digit>  
P_88a: <b integer> + <digit>  
P_88b: |<b integer> <digit>  
P_89a: <bits> + 0 <octal digit>
```
P_{90a} : \text{octal integer} \rightarrow \text{octal digit}
P_{90b} : |\text{octal integer} \text{octal digit}
P_{91a} : \text{octal digit} \rightarrow 0
P_{91b} : |\text{nonzero octal}
P_{92a} : \text{nonzero octal} \rightarrow 1
\quad .
\quad .
\quad .
\quad .
P_{92b} : | 7
P_{93a} : \text{digit} \rightarrow 0
P_{93b} : | \text{nonzero digit}
APPENDIX II: TRANSLATION RULES WRITTEN IN FORTRAN

SUBROUTINE CLEAN
ENTRY PSI1A
ENTRY PSI1C
ENTRY PSI1E
ENTRY PSI1E
ENTRY PSI2A1
K=K+1
CODELOC(K)=Z
RETURN
ENTRY PSI2A2

CODE FOR THIS NONSTOWED DECLARER IS DESTROYED AT THIS POINT. INFORMATION
ABOUT FORMAL DECLARERS IS PASSED ON.

IF (LINK(CONT(N(I)),1),ED,3RFRM) GO TO 101
ITEMP1=3RVCT
GO TO 102
101 ITEMPL=3RFRM
102 ITEMPL=IFETCH(1)
CALL STRIND (4LOREF,ITEMP)
CALL SETIND (N(I),ITEMP,2)
CALL SETIND (ITEMP1,ITEMP,1)
N(I)=ITEMP
Z=CODELOC(K)
CALL REDUCE (K)
RETURN
ENTRY PSI2B
CALL SETIND (3RVCT,N(I),1)
RETURN
ENTRY PSI3A1
ENTRY PSI3B2
A STORAGE RESERVATION COMMAND FOR THE STRUCTURE IS INSERTED HERE
BY SUBROUTINE PSI3A2

K=K+1
CODELOC(K)=Z+3
K=K+1
CODELOC(K)=Z+4
FIELDNO(K)=1

36
CALL OUTCODA2 (Z, 6HSAVEU, 10H, //)
CALL OUTCODA2 (Z, LHCURR, 10H, //)

CODE FOR THE OUTER STRUCTURE AND ITS FIRST FIELD IS INSERTED HERE:

RULE PSI3A2.

Z = Z + 2
CALL OUTCODA2 (Z, LHCURR, 10H, //)
RETURN

ENTRY PSI3A2

ALL CODE FOR FIELDS HAS BEEN GENERATED BY THIS POINT. THE REMAINING
CODE FOR THE OUTER STRUCTURE IS INSERTED HERE.

ITEM = IFETCH(1)
CALL STRING (4LISTR, ITEM)
CALL SETIND (N(I), ITEM, 2)
CALL SETIND (LINK(CONTN(I)+1), ITEM, 1)
COUNTNO = IH1E (CONTN(I), IO)
CALL PUTINF (COUNTNO, ITEM, IO)
N(I) = ITEM
CALL REDUCE (K)

CLEARS THE STACK OF INFORMATION REGARDING THE FIELDS.

INDEX = CODELOC(K)
CALL OUTCODA6 (Z, LHCURR, 2H +, CONVERT(COUNTNO, 2R), 2H +, LHCURR, 7H,
1, //)
IF (COUNTNO, GT, FIELDNO(K)) GO TO 103

IF EQUAL, THIS MEANS THAT NONE OF THE STRUCTURE FIELDS ARE STOWED,
AND HENCE, NO SUBSTRUCTURES WERE CREATED.

ITEM = CODELOC(K)-2
CALL OUTCODA1 (ITEM, 2H//)
CALL OUTCODA1 (ITEM, 2H//)
GO TO 104
103 CALL OUTCODA1 (7, 10HPOP, //)
104 CALL REDUCE (K)
RETURN

ENTRY PSI3B1
ENTRY PSI3C1
ENTRY PSI3D1

ITEM = CONVERT(R, 2RIR)
R = R + 1
IF (R, GT, MAXR) MAXR = R
CALL OUTCODA3 (Z, 3H0 +, ITEM, 10H, //)
CALL OUTCODA2 (Z, 10H, 1 + U, 10H, //)
S = S + 1
K = K + 1
CODELOC(K) = CONVERT(S + 1, 2R L)
K = K + 1
THE ROUTINE IS ALMOST PSI3A2 FOLLOWED BY PSI3D2, WITH FACILITIES FOR CARRYING OVER INFORMATION ABOUT VIRTUAL OR ACTUAL DECLARATIONS.

ITEM=IFETCH(I)
CALL STRING (4LOSTR,ITEMP)
CALL SETIND (N(I),ITEMP,2)
CALL SETINC (LINK(CONT(N(I)+1),1),ITEMP,1)
COUNTNO=BYTE(CONT(N(I)),10)
CALL PutIND (COUNTNO,ITEMP,10)
N(I)=ITEMP
CALL REDUCE (K)
INDEX=CODELOC(K)
CALL OUTCD6 (INDEX,LHCCUR,5H+,CONVERT(COUNTNO,2R),5H*,L)
105 CALL OUTCD1 (Z,10HPOP, )
106 CALL REDUCE (K)

CALL OUTCD2 (Z,CODELOC(K),10H, )
CALL OUTCD2 (Z,CODELOC(K),10H, )
CALL REDUCE (K)
CALL SETIND (N(I),N(I)-1,2)

AT THIS POINT, INFORMATION ABOUT WHETHER THE STRUCTURE IS VIRTUAL IS COMBINED WITH THE SAME INFORMATION CONCERNING THE ARRAY GENERATOR.

ITEM=LINK(CONT(N(I)),1)
ITEM=LINKCCAT(N(I)-1),1)
IF (ITEM,EQ,3RACT.AND.ITEMP1,EQ,3RACT) GO TO 108
IF (ITEM,EQ,3RFRM.OR.ITEMP1,EQ,3RFRM) GO TO 107
CALL SETIND (3RVRT,N(I)-1,1)
GO TO 108
107 CALL SETIND (3RFRM,N(I)-1,1)
108 I=I-1
R=R-1
ENTRY PSI3C

AT THIS POINT, ARRANGEMENTS ARE MADE FOR SUBSCRIPTING OF BIT AND
C: CHARACTER ARRAYS. THERE IS NO NEED TO ERASE ANY CODE FOR NONSTOWED C: DECLARERS, SINCE RULES PSI2A2, PSI4, PSI18, AND PSI19 DO THIS.

1800 IF (ITEMP .EQ. 38BIT) GO TO 110
1810 IF (ITEMP .EQ. 38AYT) GO TO 112
1819 CALL SETIND (N(I), N(I-1), 2)
1820 I=I-1
1821 RETURN
1870 INC=10H/60 + (U -
1871 R=R+1
1872 S=S+1
1873 IF (R .GT. MAXR) MAXR=R
1874 ITEMP=CONVERT(R, 2RIR)
1875 ITEMP1=CONVERT(S, 2R L)
1876 CALL OUTCDA1 (Z, IMM0, ITEMP, 10H, 16-53), 10H //)
1877 CALL OUTCDA2 (Z, ITEMP1, 16H+1 IU, , ITEMP, 9H1(56+53), INC, ITEMP, 10H1(3
1878 16-53), 10H //)
1879 CALL OUTCDA4 (Z, ITEMP, 6H +1 IU, , ITEMP, 10H, 16-53), 10H //)
1880 CALL OUTCDA5 (Z, ITEMP, 10H+DIMENSION, 1H1, ITEMP1, 10H+11 //)
1881 R=R-1
1882 GO TO 109
1887 INC=10H12+U -
1888 GO TO 111

C: ENTRY PSI3G2

C: CALL OUTCDA2 (Z, CODELOC(K), 10H, //)
C: K+K=1
C: CALL OUTCDA (Z, CODELOC(K), 10H1 //)
C: CALL REDUCE (K)
C: CALL SETIND (N(I), N(I-1), 2)
C: I=I-1
C: R=R-1
C: RETURN

C: ENTRY PSI4A1

C: K+K=1
C: CODELOC(K)=2
C: FIELDON(K)=0
C: CALL OUTCDA1 (Z, 10HSAVEU, //)
C: CALL OUTCDA2 (Z, LHCCURR, 10H+2 - U //)
C: Z=Z+1
C: CALL OUTCDA2 (Z, 9H00 - IU1, 18H //)

C: THIS OUTPUT CODE MAKES M THE INDEX OF THE DESCRIPTOR BEING CREATED.
C: A SPACE IS LEFT IN THE OUTPUT CODE FOR LATER INSERTION OF A STATEMENT
C: OF THE FORM LHCCURR+DIMENSION=LHCCURR THAT RESERVES SPACE FOR THE
C: ARRAY DESCRIPTOR ON EITHER THE HEAP OR THE LOCA STACK. RULE PSI4A2
C: MAKES THIS INSERTION
C: RETURN
C: ENTRY PSI4A2
CAUSES THE TOTAL SIZE OF THE ARRAY TO BE CALCULATED, SEES WHETHER THE
ARRAY IS VIRTUAL, AND STORES THE DIMENSION IN THE DESCRIPTOR.

BOUNDNO=LINK(FIELDNO(K),1)
I=I+1
N(I)=IFETCH(1)
CALL PUTIND(BOUNDNO,N(I),10)
CALL SETIND(3RARY,N(I),3)
VIRTUAL=LINK(FIELDNO(K),2)
IF (VIRTUAL.EQ.0) GO TO 115
IF (VIRTUAL.EQ.BOUNDNO) GO TO 113
CALL SETIND(3RFRM,N(I),1)

THE ARRAY IS FORMAL.
GO TO 114
113 CALL SETIND(3RVT,N(I),1)

THE ARRAY IS VIRTUAL, AND ITS ASSOCIATED TRANSLATION IS DESTROYED.

114 Z=CODELOC(K)
K=K-1
RETURN
115 CALL SETIND(3RACT,N(I),1)

FOR AN ACTUAL ARRAY, WE PROCEED TO PUT OUT CODE FOR COMPLETING
THE DESCRIPTOR.

ITEM=CODELOC(K)+3
ITEM1=CONVERT(BOUNDNO,2R)

WE NOW HAVE THE INFORMATION THAT PSIT81 STORED THE SIZE OF THE ARRAY
IN U AT RUN TIME; WHERE U IS CURRENTLY THE ADDRESS OF THE WORD THAT
FOLLOWS THE DESCRIPTOR IN MEMORY.

CALL OUTCDA6 (ITEM,LHCU0,R,ITEM1,4H,ITEM1,4H,5H, )
CALL REDUCE (K)
CALL OUTCDA4 (Z,3HU,-,ITEM1,8H,ITEM1,10H, )
CALL OUTCDA2 (Z,10H00,(ITEM1),10H, )
CALL OUTCDA2 (Z,10HUI(36353),10H+SIZE, )
CALL OUTCDA3 (Z,7HSIZE,0,10HU,(ITEM1),10H+17, )
CALL OUTCDA4 (Z,LHCU,R,ITEM1,10H+SIZE,ITEM1,10H+11, )
CALL OUTCDA3 (Z,ITEM1,10H+1*ITEM1,10H(36353),)

AT THIS POINT, U IS THE ADDRESS OF THE FIRST ARRAY ELEMENT.
RETURN
ENTRY PSIT5A1
ENTRY PSIT6A

CHECKS THE STRUCTURE FIELD TO DISCOVER WHETHER STOWED OR NONSTOWED, IF
NONSTOWED, DELETES PREVIOUSLY GENERATED CODE FOR POINTING FROM FIELD
TO AVAILABLE STORAGE AT RUN TIME, AND INCREMENTS A GENERAL COUNT OF
FIELDS. IF STORED OR MODE INDICANT, PUTS OUT CODE FOR INCREMENTING RUN
TIME ALLOCATION POINTER TO GENERATE A CONNECTION BETWEEN A FIELD AND
ITS DATA STRUCTURE.

IF (TYPE(I),EQ.6)RSTOWED) GO TO 116
Z=CODELOC(K)-1
FIELDNO(K)=1+FIELDNO(K)
GO TO 117
116 CODELOC(K)
CALL OUTCP2 (ITEMP,CONVERT(FIELDNO(K),2R ),10H+U*U, //)
CALL OUTCP2 (ITEMP,LHCURR,10H+[U], //)
FIELDNO(K)=1
117 CODELOC(K)=Z+1
Z=Z+2
RETURN
ENTRY PS15E1
LINKS TOGETHER TWO FIELDS OF A STRUCTURE AT COMPILE TIME. COUNTS THE
NUMBER OF FIELDS, COMBINES INFORMATION ABOUT ACTUAL AND VIRTUAL FIELDS

CALL SETIND (N(I),N(I-1),2)
CALL PUTIND (1+BYTE(CONT(N(I))),10,N(I-1),10)
ITEMP=LINK(CONT(1+N(I)),1)
ITEMP=LINK(CONT(N(I-1)),1)
IF (ITEMP,EQ.3RACT.AND.,ITEMP,EQ.3RACT) GO TO 120
IF (ITEMP,EQ.3FRFM.OR.,ITEMP,EQ.3FRFM) GO TO 118
IF (ITEMP,EQ.3RVCX.OR.,ITEMP,EQ.3RVCX) GO TO 119
CALL SETIND (3RVCX,N(I-1),1)
GO TO 120
118 CALL SETIND (3FRFM,N(I-1),1)
GO TO 120
119 CALL SETIND (3RVCX,N(I-1),1)
120 I=I-1
RETURN
ENTRY PS17A1
CALL SETDIF (1,FIELDNO(K),1)
CALL OUTCDA2 (Z,10H01+U11136*+10H53), //)
CALL OUTCDA2 (Z,10H01+U11115*35,10H-[(U)(0-17),6*SIZE,10H-1+U, //
1)
CALL OUTCDA4 (Z,5H0*U,,7*SIZE>01,10H_SIZE=EU)(3,10H6*53)11 //)
RETURN
ENTRY PS17B1
C COUNTS UP THE NUMBER OF DIMENSIONS, STORES THE STRIDE IN THE CURRENT
C DESCRIPTOR WORD, AND zeroes THE NEXT DESCRIPTOR WORD.
C CALL SETDIF (1+LINK(FIELDNO(K),1),FIELDNO(K),1)
CALL OUTCDA3 (Z,10H0118*35,10H-[U](0-17),10H SIZE, //)
C CALCULATES UPPERBOUND-LOWERBOUND.
CALL OUTCD3 (7,10H$IZE*U(3,10H$653)*SIZE,10H, //)
CALL OUTCD4 (7,10H$IZE*U, //)
    7=2+1
CALL OUTCD4 (7,6H$IZE*U,7H$IZE*01,10H$IZE*U(3,10H$653)!! //)
RETURN
ENTRY PSI3A

GENERATES A CELL IN THE Compile-TIME LIST COPY OF A STRUCTURE DECLARATION.
ITEMP=IFETCH(2)
CALL STRING(N(I),ITEMP+1)
I=I-1
IF (TYPE(I),EQ,6RSTONE) GO TO 121
CALL STRING (4L0+FL,ITEMP)
GO TO 122
121 CALL STRING (4L1+FL,ITEMP)
122 CALL SETINC (N(I),ITEMP,2)
N(I)=ITEMP
RETURN
ENTRY PSI9A
I=I+1
N(I)=NAME
RETURN
ENTRY PSI10A

THIS ROUTINE COUNTS VIRTUAL BOUNDS IN AN ARRAY DECLARATION.
CALL SETDIR (1*LINKFIELDNO(K),2),FIELDNO(K),2)

NOTE THAT ANYTHING BUT ZERO IN THE SECOND FIELD OF FIELDNO INDICATES THAT THE ARRAY IS NOT AN ACTUAL ARRAY.
RETURN
ENTRY PSI11A
ITEMP1=10HU1(55+55)
ITEMP2=10HU1(0+17)
GO TO 123
ENTRY PSI12A
ITEMP1=10HU1(54+54)
ITEMP2=10HU1(18+35)
123 GO TO (124,126,128,125,125), BOUND
124 CALL OUTCD3 (Z,5H 01*,ITEMP,10H, //)
125 CALL OUTCD4 (7,CONVERT(P,2R T),14*,ITEMP2,18H, //)
P=P+1
I=I-1
126 RETURN
    ENTRY PSI13B
    SOUND=2
    RETURN
ENTRY PSI13C
    SOUND=3
    RETURN
ENTRY PSI13A
    SOUND=1
    GO TO 127
ENTRY PSI13D
    SOUND=4
    GO TO 127
ENTRY PSI13E
    SOUND=5
127 IF (SAME TF(N(I), INTEGER, REF)) GO TO 128
    CALL ERROR(110, NON-INTEGER ARRAY BOUND IN ROUNOISLIST.)
1 RETURN
128 CALL OUTEVAL (P, REF, OUTCODE, Z, 1000-Z)
RETURN
    ENTRY PSI14C
    ITEM=3RREA
    GO TO 130
ENTRY PSI14D
    ITEM=3RINT
    GO TO 130
ENTRY PSI14E
    ITEM=3RALT
    GO TO 130
ENTRY PSI14F
    ITEM=3RBYT
    GO TO 130
ENTRY PSI15A
129 ITEM=3OPRC
130 I=I+1
  ITEMD=IFETCH(1)
  CALL SETIND (ITEMD,ITEMD,3)
  RETURN
ENTRY PSI15B1
K=K+1
CODELOC(K)=7
GO TO 129
ENTRY PSI15B2

CODE GENERATED DURING ANALYSIS OF THE <VIRTUAL PROCPLAN> IS DESTROYED.
Z=CODELOC(K)
CALL REDUCE (K)
RETURN
ENTRY PSI16A
CALL PUTINC (1+IBYTE(CONT(N(I)),10),N(I),10)
ITEMD=2
GO TO 131
ENTRY PSI16B

ITEMD=1
131 ITEMP=IFETCH(1)
CALL SETIND (N(I),ITEMP,ITEMD)
I=I-1
CALL SETINC (ITEMD,N(I),2)
RETURN
ENTRY PSI16C
ITEMD=IFETCH(1)
CALL SETINC (N(I),ITEMD,1)
CALL SETIND (N(I-1),ITEMD,2)
CALL PUTINC (IBYTE(CONT(N(I-1)),10),N(I),17)
I=I-2
N(I)=ITEMD
RETURN
ENTRY PSI17A
ITEMD=1
GO TO 132
ENTRY PSI17B1
ITEMD=0
132 ITEMP=IFETCH(1)
CALL SETIND (3PRM,ITEMP,3)
CALL SETINC (N(I),ITEMD,2)
CALL PUTINC (ITEMP1,ITEMP,10)
N(I)=ITEMP
RETURN

ENTRY PSI1782

THIS ROUTINE LINKS TOGETHER THE PARAMETERS IN THE COMPILER COPY OF THE
PARAMETER LIST, AND COUNTS THE NUMBER OF PARAMETERS.

CALL PUTINO (1+BYTECONT(N(I)),101,N(I-1),10)
I=I-1
RETURN

ENTRY PSI18A1

K=K+1
CODELOC(K)=Z
RETURN

ENTRY PSI19A2

?=CODELOC(K)
CALL REDUCE (K)
RETURN

ENTRY PSI19A

ENTRY PSI19B1

ITEMP=IFETCH(1)
CALL SETIND (3RUNI,ITEMP,3)
CALL SETIND (N(I),ITEMP,2)
N(I)=ITEMP
RETURN

ENTRY PSI19B2

CALL SETIND (N(I),N(I-1),1)
I=I-1
RETURN

ENTRY PSI20A

I=I+1
ITEMP=IFETCH(2)
N(I)=ITEMP
CALL STRING (NAME,ITEMP+1)
CALL SETIND (3MOD,IEMT,3)
NAME=ALIAS(NAME)

THE ALIAS FUNCTION LOOKS UP THE RUN-TIME PROGRAM LABEL ASSIGNED TO THE
MODE INDICANT, AND ADDS THE INDICANT TO THE COMPILER DECLARATION TABLE
IF NECESSARY.
CALL OUTCDA4(Z,10HSAVE RETURN,2HN,10HJ+2=RETURN,3H,)
CALL OUTCDA4(Z,NAME,2H,,10HPOP RETURN,10H,)
ENTRY PSI21A
A CHECK IS MADE HERE TO SEE WHETHER THE DECLARER IN QUESTION CAN BE
USED AS A VIRTUAL DECLARER, NO CHECK IS NEEDED FOR FORMAL DECLARERS.
ITEM=LINKCONT(I),I
IF (ITEM.EQ.3RVRT.OR.ITEM.EQ.3RVCT) RETURN
CALL ERROR(110HAN ATTEMPT WAS MADE TO USE A FORMAL OR ACTUAL O
DECLARER IN PLACE OF A VIRTUAL DECLARER.)
RETURN
ENTRY PSI22A
A CHECK IS MADE TO SEE WHETHER THE DECLARER IN QUESTION CAN BE USED AS
AN ACTUAL DECLARER, NO CHECK IS NEEDED FOR FORMAL DECLARERS.
ITEM=LINKCONT(I),I
IF (ITEM.EQ.3RVCT.OR.ITEM.EQ.3SVCT) RETURN
CALL ERROR(110HAN ATTEMPT WAS MADE TO USE A FORMAL OR VIRTUAL
DECLARER IN PLACE OF AN ACTUAL DECLARER.)
RETURN
ENTRY PSI25A1
ENTRY PSI26A1
K=K+1
CODELOC(K)=2
THIS RULE ALLOWS DESTRUCTION OF CODE PRODUCED BY RULE PSI20A.
RETURN
ENTRY PSI25A2
Z=CODELOC(K)+2
A SPACE IS RESERVED FOR A TRANSFER OF CONTROL AROUND THE MODE
DECLARATION IN THE TRANSLATED PROGRAM.
CALL OUTCDA2(Z,NAME,10H1,)
THE MODE INDICANT WAS DECLARED BY PSI20A, AND NAME CONTAINS THE RUN-
TIME LABEL ASSOCIATED WITH THE INDICANT.
FIELDN0(K)=MODE
MODE IS THE INDEX OF THE COMPILER MODES TABLE.
RETURN
ENTRY PSI25A3

CALL OUTCOA3 (Z,9HRETURN=RSJ,8H(30#47),,1OHRS, , //)
S=S+1
ITEMP=CONVERT(S,2R L)
ITEMP=CODELOC(K)+1
CALL OUTC0B2 (ITEMP,ITEMP1,1OH, , //)
CALL OUTC0A2 (2,ITEMP1,1OH; //)
ITEMP=FIELDNO(K)
MODES(ITEMP)="N(I)

ASSIGNS COMPILER DATA STRUCTURE TO MODE INDICANT.

N=I-1
CALL REDUCE (K)
RETURN

ENTRY PSI26A2

AT THIS POINT, THE OPERATOR HAS BEEN ENTERED INTO THE OPS TABLE. THE
CURRENT INDEX OF OPS IS OP, AND THE RUN-TIME ALIAS OF THE OPERATOR IS
STORED IN THE COMPILER VARIABLE NAME, EXACTLY AS FOR THE CASE OF MODE
INDICANTS,

Z=CODELOC(K)+2

A SPACE IS RESERVED FOR A TRANSFER OF CONTROL AROUND THE OPERATOR
DEFINITION IN THE TRANSLATED PROGRAM.

CALL OUTC0B2 (Z,NAME,10H; //)
FIELDNO(K)=OP
RETURN

ENTRY PSI26A3

CALL OUTCOA3 (Z,9HRETURN=RSJ,8H(30#47),,1OHRS, , //)
S=S+1
ITEMP=CONVERT(S,2R L)
ITEMP=CODELOC(K)+1
CALL OUTC0B2 (ITEMP,ITEMP1,1OH, , //)
CALL OUTC0A2 (2,ITEMP1,1OH; //)
ITEMP=FIELDNO(K)
M(CODEPOS)=N(I)

THIS ROUTINE ACHIEVES AN EXTRA LEVEL OF REFERENCING TO THE DECLARER, BOTH IN
COMPILED REPRESENTATION AND THE RUN-TIME CODE. LHCURR CARRIES INFORMA
ABOUT WHETHER LOG OR HEAP IS IN USE.

S=S+1
ITEMP=CONVERT(S,2R L)

ENTRY PSI27A

ENTRY PSI28A

ENTRY PSI30A
FIELDNO(K)=ITEMP
ITEMP1=CODELOC(K)+1
CALL OUTCOPE (ITEMP1,ITEMP,4H,1+,LHCURR,1H,+,LHCURR,8H, //)
ITEMP1=ITEMP1+2
S=S+1
ITEMP=CONV (S,2R L)
CALL OUTCDA (ITEMP,ITEMP,10H, //)
CALL OUTCDA (2,ITEMP,10H, //)
ITEMP=IFETCH (1)
CALL SETIND (3,REF,ITEMP,3)
CALL SETIND (N(I),ITEMP,2)
N(I)=ITEMP
RETURN
ENTRY PSI27A
LIKE PSI27A, EXCEPT THAT THE EXTRA LEVEL OF RUN-TIME INDIRECTNESS IS
SUPPLIED BY PSI3DA1.

S=S+1
ITEMP=CONV (S,2R L)
FIELDNO (K)=ITEMP
ITEMP1=CODELOC (K)+1
CALL OUTCOPE (ITEMP1,ITEMP,10H, //)
S=S+1
ITEMP=ITEMP1+1
ITEMP=CONV (S,2R L)
GO TO 133
ENTRY PSI27C1
THE TAG RULE PSI68A HAS THE EFFECT OF ENTERING THE NAME ONTO THE NAMES
TABLE, AND LEAVES THE RUN-TIME ALIAS OF <TAG> IN THE VARIABLE NAME.
CODE FOR THE VIRTUAL DECLARER IS DESTROYED

OPERAND(I)=NAME
TYPE(I)=9RNONSTORED
Z=CODELOC(K)
RETURN
ENTRY PSI27C2
IF (SAMEYFPN(I=I+1,N(I),REF),AND,REF,EQ,0) GO TO 134
CALL ERROR (110,ATTEMPT TO INITIALIZE A DECLARED VARIABLE
USING A <TERTIARY> OF A DIFFERENT MODE.)
RETURN
134 IF (TYPE(I),NE,9RNONSTORED) GO TO 135
P=P+1
ITEMP=CONV (P,2R T)
CALL OUTCDA (Z,OPERAND(I),1H,+,ITEMP,10H, //)
OPERAND(I)=ITEMP
135 I=I-1
OPERAND(I)=IHASHB(OPERAND(I),N(I))

SETS THE TYPE OF THE NAME AND THE REGION WHERE ITS VALUE IS STORED.

CALL OUTCDA4(2,OPERAND(I+1),1H*,OPERAND(I),10H, //)
RETURN

ENTRY PSI28A
ENTRY PSI28C

ENTER THE <TAG> AND ITS MODE ONTO THE NAME TABLE.

NAME=IHASHB(NAME,N(I))
CALL OUTCDA4 (2,LHICURR,4H+1*E,NAME,10H) //
CALL OUTCDA3 (2,10HSAVE RETUR,10HN,J+2=RETU,10HN, //
CALL OUTCDA2 (2,FIELDNO(K),10H, //
CALL OUTCDA2 (2,10HPOP RETURN,10H, //
ENTRY PSI28B1
ENTRY PSI28D1

OPERAND(I)=IHASHB(NAME,N(I))
TYPE(I)=9RNONSTORRED
RETURN

ENTRY PSI28B2
ENTRY PSI28D2

THE DATA TYPES OF <TERTIARY> AND <TAG> ARE COMPARED. THE <TERTIARY> IS REFERENCED AND DEPROCEDURED TO ONE LEVEL OF REFERENCE BELOW N(I-1)

NOTE THAT THE FIRST PARAMETER OF SAMEPYT IS THE ONE TO BE LOWERED.

IF (SAMEPYT(N(I)),LINK(CONT(N(I-1)),1),REF)) GO TO 169
CALL ERROR (110-<THE <TERTIARY> ASSIGNED TO THE <TAG> OF THIS MODE DECLARATION DOES NOT POSSESS THE SAME MODE AS ITS ASSIGNEE.>)
RETURN
169 IF (TYPE(I),NE.9RNONSTORRED) GO TO 170
P=P+1
ITEM=CONVERT(P,2R 1)
CALL OUTCDA4 (2,OPERAND(I),1H*,ITEM,10H, //
170 CALL OUTCVA3 (P,REF,OUTCODE,7,1000-Z)
I=I+1
CALL OUTCDA3 (2,ITEM,1H*,OPERAND(I),10H, //
RETURN

ENTRY PSI29A1
ITEM=IFETCH(1)
CALL SETINO (3*REF,ITEM,3)
CALL SETINO (N(I),ITEM,2)
N(I)=ITEM
RETURN

ENTRY PSI30A1
L4CURR=4HHEAP

HERE WE PUT OUR CODE FOR GENERATING AN EXTRA LEVEL OF REFERENCING AT
RUN TIME. THE BEGINNING IS SAVED FOR USE AS DESTINATION IN ASSIGNMENTS

CALL OUTCOA5 (7,10HHEAP+1+HEA,10HP,HEAP+1 10HHEAP), 8HSAVEU, 10HHEAP+U, //
RETURN

ENTRY PSI30A2
L4CURR=3HLCC
CALL OUTCOA3 (7,10H U=GENERAT,10HOR VALUE, 10HP POP U, //
THE VALUE OF THE GENERATOR IS PRESERVED FOR USE IN ASSIGNMENTS9
OUTCOE(7,?)=10HP POP U, //
RETURN
ENTRY PSI31A1
CALL INITIAL(NAME,MODE)
ENTRY PSI31A2
CALL FINAL(NAME,MODE)
RETURN
ENTRY PSI33A1
ENTRY PSI33B1
L4CURR=3HLCC
CALL BLOK9GN(P,MODE)
ITEMP=Z+3 8210
K=K+1 8220
CODELOC(K)=ITEMP 8230
K+K+1 8240
CODELOC(K)=ITEMP 8250
RESERVES SPACE FOR THE TRANSLATION OF INITIALIZATION DECLARATIONS AND
FOR MANAGING RUN-TIME STORAGE ALLOCATION.
Z.Z+4
RETURN
ENTRY PSI33A2
ENTRY PSI33B2
CALL BLOKEND(P,MODE)
RETURN

8100
8110
8120
8130
8140
8150
8200
8210
8220
8230
8240
8250
8260
8270
8280
8290
8300
8310
8320
8330
8340
8350
8360
8430
8440
PSI348 CAUSES LOCAL BLOCK STORAGE TO BE ERASED IN THE TRANSLATED
PROGRAM.

ENTRY PSI34A

FILLS IN BLANK SPACES AT THE BEGINNING OF TRANSLATED CODE FOR THE
CURRENT BLOCK.

ITEM=CODELOC(K)+2
ITEM1=ITEM+3
ND 13A ITEM2=ITEM,ITEM1,1
13A OUTCODE(ITEM2,1)=2H//
K=K-1
RETURN

ENTRY PSI34B1

K=K-1
RETURN

ENTRY PSI34B2

CODE IS PUT OUT AT THIS POINT TO FILL IN THE STORAGE MANIPULATION
BLANKS LEFT BY PSI33A1 AND PSI33A1.

ITEM=CODELOC(K)+2
"ALL OUTCOD2 (ITEM,10H1+RLOKNUM,10H1ROKNUM, //)
CALL OUTCOD3 (ITEM,4WLOC1,10H1STORAGE1BL,13HOKNUM, //)
CALL OUTCOD6 (Z,10H1STORAGE1BL,13HOKNUM1=LOC,1H,10H1ROKNUM-1,10H1BL
10KNUM,10H //)
RETURN

ENTRY PSI36A

CLEARS THE MOST RECENT DECLARATION FROM THE N STACK AND RESETS CODELOC
FOR THE NEXT DECLARATION.

L4CURR=3HLCC
CODELOC(K)=Z+1
Z=Z+2
I=I-1
RETURN

CASE/PSI33A1/<SER. CL.>IN/PSI32D2/<ROW OF CL.>FI/PSI32G3/
CASE/PSI33A1/<SER. CL.>IN/PSI32D2/<ROW OF CL.>ELSE/PSI32D3/<SER. CL.>FI

ENTRY PSI32D2

IF(.NOT.SAMEYP(N,I),INTREAL,REF)
1 CALL ERROR(10M1INDEX OF A CASE STATEMENT IS NOT A NUMERICAL QU
2ANTITY.
)
IF (.NOT.(TYPE(I).EQ.9RNONSTOR)) GO TO 183
P=P+1
ITEM=CONVEPT(P,2R T)
CALL OUTCOMA \( (Z, \text{OPERAND}(I), 1H, \text{ITEMP}, 10H, \quad //) \)
CALL OUTVAL \( (P, \text{REF}, \text{OUTCODE}, Z, 1000-Z) \)
I=I+1
CODELOC(K)=7+1
CALL OUTCOMA \( (Z, \text{ITEMP}, 6H \leq 01, 01, 10H, \quad //) \)
CALL OUTCOMA \( (Z, \text{ITEMP}, 6H > 01, 01, 10H, \quad //) \)
CALL OUTCOMA \( (Z, \text{ITEMP}, 6H + 01, 01, \text{ITEMP}, 10H, \quad //) \)
CALL OUTCOMA \( (Z, \text{ITEMP}, 10H + RJ(30+47), 10H, RJ,), \quad //) \)

T(P)<0, ELSE ::
T(P)> NUMBER OF CLAUSES IN ROWI LABEL ELSE ::
T(P)+ LABEL FI = T(P), (SEE PSI8203 FOR COMPLETION.)
T(P)-RJ(30+47), RJ.,
CAUSES A JUMP TO A TABLE LEADING TO THE APPROPRIATE CLAUSE OF THE
CASE STATEMENT, NOTE THAT STORAGE (BLOKNUM) IS USED TO RESET P TO THE
SAME LEVEL FOR EACH CLAUSE.
THE SAVING OF CODELOC(K) IN WHAT FOLLOWS IS PERFORMED FOR COMPATIBILI-
WITH PSI33A1 WHEN A<ROW OF CL.> APPEARS WITHIN A <COLLATERAL CLAUSE>.

CALL BLOKEND(P,MODE)
K=K*1
CODELOC(K)=Z
9220
9230
9240
9250
9260
9270
9280
9290

ENTRY PSI8203 .

SEARCH HERE FOR THE MOST DEREFERENCED AND DEPROCEDURED (I.E., LOWEST)
CLAUSE IN THE <ROW OF CLAUSE>, AND BRING THE REMAINING CLAUSES DOWN
TO THAT LEVEL.

CALL BLOKEND(P,MODE)
CALL BLOKEND(P,MODE)
K=K*2
S=S-1
Z=Z-1
LINK1=MINIMUM(N(l))
9330
9340
9350
9360
9370

RULES PSI183A AND PSI136 FOR <ROW OF CLAUSE> CREATE A LIST OF LABELS
ON TOP OF THE OPERAND STACK, EACH ELT. OF THE LIST CONTAINS IN SEQUENC
(1) A POINTER TO THE NEXT ELT.
(2) THE LABEL WHERE THE UNITARY CLAUSE BEGINS.
(3) THE LABEL WHERE DEREFERENCING AND DEPROCEDURING
OF THAT CLAUSE OCCURS.
(4) THE LOCATION IN THE CODE OF (3) WHERE ESCAPE IS
MADE TO THE CODE DIRECTLY FOLLOWING THE CASE
STATEMENT, (SEE BALANCE SUBROUTINE.)

ITEMP1=OPERAND(I)
ITEMP2=N(I)
CALL BALANCE \( (\text{LINK1}, \text{ITEMP1}, \text{ITEMP2}, S, P, Z, \text{OUTCODE}) \)
N(I)=LINK1
9390
9400
9410
9420
9430
9440
9450
9460
9470
9480
9490
9500
9510
9520
9530
9540
LINK1 POINTS TO THE LOWEST DATA TYPE IN THE ROW OF CLAUSE.
SEE PSIA202 FOR CONTENTS OF CODELOC(K) AND CODELOC(K-1) HERE.

S=S+1
ITEM=CONVERT(S,2R L)
CALL OUTCD2 (Z,ITEM,10H1, //)
ITEM1=CODELOC(K)
K=K-1
OUTCODE(ITEM1+2,3)=ITEM

THE LOCATION OF THE CASE STATEMENT JUMP TABLE IS PUT INTO THE JUMP COD

ITEM2=OPERAND(I)
139 CALL OUTCD2 (Z,CONT(ITEM2+2),10H*, //)
ITEM2=LINK(CONT(ITEM2),1)
IF (ITEM2 GT 0) GO TO 139

THE JUMP TABLE IS GENERATED, THE ELSE CLAUSE LABEL IS NEXT INSERTED.

S=S+1
ITEM=CONVERT(S,2R L)
CALL OUTCD2 (Z,ITEM,10H1 //)
OUTCODE(ITEM1,3)=ITEM
OUTCODE(ITEM1+1,3)=CONVERT(1BYTE(CONT(OPERAND(I)),10),2R )
OUTCODE(ITEM1+1,5)=ITEM
GO TO 184

ENTRY PSIA204

FURTHER BALANCING TAKES PLACE HERE BETWEEN THE <ROW OF CLAUSE> AND TH
<SERIAL CLAUSE>.
AFTER SAVING THE BLOCK VALUE, DO THE BLOCK END BOOKKEEPING.

CALL RLOEND( P, MODE)
O=P+1
ITEM4=CONVERT(P,2R T)
IF (OPERAND(I),EQ.ITEM4) GO TO 140

CALL OUTCD4 (Z,OPERAND(I),1H,ITEM4,10H, //)
140 K=K-1

TEST HERE FOR WHETHER THE <SER. CL.> IS LOWER THAN THE <ROW OF CL.>
IF LOWER, JUMP AT RUN TIME TO THE CODE FOLLOWING THE CASE STATEMENT,
CAUSE THE <ROW OF CL.> RESULTING TO BE FURTHER REFERENCED AND DEPROC
IF HIGHER, DEREFERENCE AND DEPROCEDURE THE <SER. CL.>; THEN INSERT TH
FOR JUMPING OUT OF THE <ROW OF CL.> ROUTINES.

IF (.NOT.SAMETYPE(N(I-1),N(I),REF)) GO TO 141
IF (REF,EQ.0) GO TO 142
GO TO 144
141 IF (SAMETYPE(N,I,N(I),REF)) GO TO 145

CALL ERROR(110)THE REMAINING CLAUSE AFTER*ELSE* IN THE CASE ST
STATEMENT DOES NOT HAVE THE SAME MODE AS PRECEDING CLAUSES.
)
RETURN
NO BALANCING IS NEEDED HERE.

142 $= S + 1$
ITEMP=CONVERT($S, 2R L$)
CALL OUTCDA2 ($Z, ITEMP, 10H1$)
CALL LOOP82D (OPERAND(I-1), ITEMP, OUTCODE)

Provides escape labels to the case clauses.

143 I=I-1
TYPE(I)=6RSTORED
OPERAND(I)=ITEMP
RETURN

The entire <row of CL>, after being balanced internally, is here balan
the serial else clause.

144 $S = S + 1$
ITEMP=CONVERT($S, 2R L$)
CALL OUTCDA2 ($Z, ITEMP, 10H1$, 10H$)
ICYEP(I)=ITEMP
S=S+1
ITEMP=CONVERT($S, 2R L$)

Do LOOP82D to force all <row of CL> to escape to this point in the
control. Note that value return is used to carry the values of each cla

CALL LOOP82D (OPERAND(I-1), ITEMP, OUTCODE)

CALL OUTEVAL (P, REF, OUTCODE, Z, 1000-Z)
CALL OUTCDA2 ($Z, TYPE(I), 10H1$, 10H$)
M(I-1)=N(I)
GO TO 143

The <serial CL> must be dereferenced and deprocedured to balance with
<row of CL>.

145 $S = S + 1$
ITEMP=CONVERT($S, 2R L$)
CALL OUTCDA2 ($Z, ITEMP, 10H1$, 10H$)
CALL OUTEVAL (P, REF, OUTCODE, Z, 1000-Z)
CALL LOOP82D (OPERAND(I-1), ITEMP, OUTCODE)
CALL OUTCDA2 ($Z, ITEMP, 10H1$, 10H$)
GO TO 143

ENTRY PS182C3
CALL BLOKEND(P, MODE)
K=K-2
LINK1=MINIMUM(N(I))
S=S-1
Z=Z-1
CALL BALANCE (LINK1, OPERAND(I), N(I), $S, P, Z, OUTCODE$)
N(I)=LINK1
TYPE(I)=6RSTORED
S=S+1
ITEMP=CONVERT(S,2R L)
CALL OUTCDA2 (Z,ITEMP,10H,       //
ITEMP=CODELOC(K)
K=K+1
OUTCODE(ITEMP+2,3)=ITEMP
ITEMP=OPERAND(I)
ITEMP3=ITEMP+3
146 CALL OUTCDA2 (Z,CONT(ITEMP3+2),10H,       //
ITEMP3=CONVERT(ITEMP3+1)
IF (ITEMP3>0) GO TO 146
S=S+1
ITEMP=CONVERT(S,2R L)
CALL OUTCDA2 (Z,ITEMP,10H,       //
OUTCODE(ITEMP,3)=ITEMP
ITEMP=ITEMP+1
OUTCODE(ITEMP,3)=CONVERT(INTYTE(CONT(OPERAND(I)),10),2R)
OUTCODE(ITEMP,5)=ITEMP
P=P+1
ITEMP=CONVERT(P,2R T)
CALL OUTCDA3 (Z,SH0 +,ITEMP,10H,       //
S=S+1
ITEMP=CONVERT(S,2R L)
CALL OUTCDA2 (Z,ITEMP,10H,       //
CALL LOOP2D (OPERAND(I),ITEMP,OUTCODE)
TYPE(I)=6RSTORED
OPERAND(I)=ITEMP
RETURN
IF/PSI33A1/<SERIAL CL.>THEN/PSI82A2/<SER.CL.>ELSE/PSI82A3/<SER.CL.>FI
IF/PSI33A1/<SERIAL CL.>THEN/PSI82A2/<SER.CL.>FI/PSI82B3/ENTRY PSI82A2
IF (NOT,SAMEYP(N(I),BOOL,REF))
1 CALL ERROR(110HIF STATEMENT WITH NON-BOOLEAN TEST.
2 )
IF (NOT,(TYPE(I),EQ,9RNONSTORED)) GO TO 147
P=P+1
ITEMP=CONVERT(P,2R T)
11060
147 CALL OUTCDA4 (Z,OPERAND(I),1H,ITEMP,10H,       //
I=I+1
CALL OUTCDA4 (Z,CONVERT(P,2R T),3H=01,0,10H,       //
CODELOC(K)=Z
RETURN
: CODELOC CARRIES LOCATION TO INSERT FIXUP LABEL HERE.
148 CALL BLOKEND(P,MODE)
   CALL BLOKBGN(P,MODE)
ITEMP=Z+3
K=K+1
CODELOC(K)=ITEMP
K=K+1
CODELOC(K)=ITEMP
Z=ITEMP+1
RETURN

ENTRY PSIB2A3
IF IOPFRAND(I)=EQA.ITEMP) GO TO 149
ITEMP=CONVERT(IBYTE( items=0),10H,ITEMP=0H,10H, 10, 10H, 10H, //)
TYPE(I)=6RSTORED
OPERAND(I)=ITEMP
149 CALL OUTCDA2 (7,0,10H, //)
INSERT SKIP CLAUSE LABEL HERE.

K=K-1
ITEMP1=CODELOC(K)
CODELOC(K)=7
S=S+1
ITEMP=CONVERT(S(2R L)
CALL OUTCDA2 (7,ITEMP,10H, //)
OUTCODE(ITEMP1,3)=ITEMP
GO TO 148

ENTRY PSIB2A4
CALL BLOKEND(P,MODE)
P=P+1
ITEMP=CONVERT(P,2R T)
IF (OPERAND(I)=EQ,ITEMP) GO TO 150
CALL OUTCDA4 (2,OPERAND(I),10H,ITEMP,10H, //)
150 K=K-1
BALANCING OF CONSEQUENCE AND ALTERNATIVE1
IF (.NOT.SAMEYF(N(I-1),N(I),REF)) GO TO 151
IF (REF,EQ,0) GO TO 152
GO TO 153
151 IF (SAMEYF(N(I),N(I-1),REF)) GO TO 155
CALL ERROR110THE CONSEQUENCE AND ALTERNATIVE OF AN IF STATEMEN
1ENT DO NOT HAVE THE SAME A POSTERIORI MODE.
11570
11590
11600
11610
11620
11630
11640
11650
11660
11670
11680
11690
11700
11710
11720
11730
11740
11750
11760
ITEM1=CONVERT(S,2R L)
ITEM2=CODELOC(K)
K=K+1
CALL OUTCD2 (Z,ITEM1,10H) //
OUTCODE(ITEM2,1)=ITEM1
CALL OUTEVAL (P,REF,OUTCODE,Z,1000-Z)
S=S+1
ITEM1=CONVERT(S,2R L)
CALL OUTCD2 (Z,ITEM1,10H) //
ITEM2=CODELOC(K)
K=K+1
OUTCODE(ITEM2,1)=ITEM1
GO TO 154

I=I-1

ENTRY PSI82B3

INSERT ALTERNATIVE CODE AND PROVIDE A JUMP TO AVOID THIS CODE.
ITEM=STORAGE (BLOKNUM)
P=BYTE (ITEM1,10)+1
ITEM1=CONVERT(P,2R T)
CALL BLOKEND (0,0)
CALL BLOKSGN (0,0)
IF (OPERAND(1).EQ.ITEM1) GO TO 156
CALL OUTCD2 (Z,OPERAND(1),1H*,ITEM1,10H, //)
YPF(I)=6RSTORED
OPERAND(I)=ITEM1

156 CALL OUTCD2 (Z,0,10H, //)
S=S+1
ITEM2=CONVERT(S,2R L)
K=K+1
ITEM2=CODELOC(K)
K=K+1
CALL OUTCD2 (Z,ITEM2,10H) //
OUTCODE(ITEM1,3)=ITEM2
CALL OUTCD2 (Z,ITEM2,10H, //)
S=S+1
ITEM2=CONVERT(S,2R L)
OUTCODE(7-2,1)=ITEM2
CALL OUTCD2 (Z,ITEM2,10H) //
RETURN
ENTRY PSI84A

C TEST TO SEE WHETHER WE ARE COMPILING THE FIRST CLAUSE IN A ROW OF CL
C THEN, IF TRUE, INSERT A JUMP AROUND THE ROW OF CLAUSE TO WHERE THE EVAL
JUMP TABLE BEGINS.

P=BYTE (STORAGE (BLOCKNUM),10)
ITEM=CONVERT (P+1,2R T)
IF (OPERAND (I),EQ,ITEM) GO TO 157
CALL OUTCOD4 (Z,OPERAND (I),1H,ITEM,10H, / /
157 CALL OUTCOD3 (Z,10HRJ, RETURN + RJ,8H(3047), 10HRJ, / /
ITEM=IFETCH (4)
IF (CODELOC (K),EQ,-1) GO TO 158

THIS TEST DISTINGUISHES THE FIRST CLAUSE OF THE ROW.

K=K+1
CODELOC (K)=-1
S=S+1
ITEM3=CONVERT (S,2R L)

THE DOUBLE USE OF <ROW OF CL.> WITHIN CASE STATEMENTS AND COLLATERAL
REQUIRES SOME BLANKING OF OUTCODE CARDS PUT IN BY PSI33A1 DURING PSI3A

ITEM2=CODELOC (K-2)
CALL OUTCOD2 (ITEM2,ITEM3,10H, / /
CODELOC (K-1)=ITEM3
CODELOC (K-2)=ITEM2-1
158 CALL STPIND (CODELOC (K-1),ITEMP1+1)
OPERAND (I)=ITEMP1

STORED ON OPERAND (I) IN A LIST CELL.

S=S+1
ITEM3=CONVERT (S,2R L)
CODELOC (K)=ITEMP3
CALL OUTCOD2 (Z,ITEM3,10H, / /
RETURN

THE LABEL AT WHICH CODE BEGINS FOR EACH ROW IN A <ROW OF CLAUSE> IS

ENTRY PSI143A

DO PSI144A WORK FIRST, THEN PIECE TOGETHER THE COMPILE-TIME REPRESENTAT
OF <ROW OF CLAUSE>.

P=BYTE (STORAGE (BLOCKNUM),10)
ITEM=CONVERT (P+1,2R T)
IF (OPERAND (I),EQ,ITEM) GO TO 159
CALL OUTCOD4 (Z,OPERAND (I),1H,ITEM,10H, / /
159 CALL OUTCOD3 (Z,10HRJ, RETURN + RJ,8H(3047), 10HRJ, / /
ITEM=IFETCH (4)
CALL STRIND (CODELOC (K-1),ITEMP1+1)

C CONSTRUCT LIST OF <ROW OF CL.> PROCEDURES.
C
CALL STRIND (ITEM, OPERAND (I-1))
C
C CONSTRUCT A COMPILE TIME STRUCTURE SKELETON REPRESENTATION OF THE DATA

ITEMP=IFETCH(1)
CALL SETNU(N(I),ITEMP,2)
I=I-1
ITEMP=IFETCH(1)
CALL SETIND(N(I),ITEMP,1)
CALL SETNE(ITEMP,ITEMP,1)
CALL PUTNE(2,ITEMP,10)
N(I)=ITEMP
RETURN

ENTRY PS1838

ITEMP=IFETCH(1)
CALL SETNU(N(I),ITEMP,1)
I=I-1
CALL SETIND(N(I),ITEMP,2)
N(I)=ITEMP
CALL PUTNE(1+IBYTE(CONT(N(I+1)),10),ITEMP,10)
CALL STIND(OPERAND(I+1),OPERAND(I))
RETURN
ENTRY PS185A2

INSERT JUMP ARUGO <POW OF CLAUSE> CODE THEN GO TO BLOCKEND ROUTINE.

K=K-2
ITEMP=CODELOC(K)-2
ITEMP1=ITEMP+1
DO 160 I=ITEMP,ITEMP1
160 OUTCODE(I,I)=2H/
   CALL OUTCODE2(CODELOC(K),CODELOC(K+1),10H,)
   GO TO 164

ENTRY PS14802

AT THIS POINT, THE MODE OF THE <VIRT. MODE DECLARER> MUST BE EITHER S
OR ARRAY, WITH NO PROC OR REF PREFIXES. IN ADDITION, THE <COLLATERAL
AND ITS A POSTERIORI MODE MUST BE THE SAME LENGTH.

ITEMP1=CONT(N(I))
ITEMP2=CONT(N(I-1))
P=P+1
ITEMP3=IBYTE(ITEMP1,10)
IF (ITEMP3.EQ.1BYTE(ITEMP2,10)) GO TO 161
CALL ERROR(110)THE <COLLATERAL CLAUSE> AND ITS A POSTERIORI MODE
IS NOT THE SAME LENGTH.
RETURN
161 ITEMP4=LINK(ITEMP1,3)
IF (ITEMP4.EQ.3RARY) GO TO 162
IF (ITEMP4.EQ.3STR) GO TO 164
CALL ERROR(110)THE A POSTERIORI MODE OF A <MOID CAST> WITH A <CO
1LATEPAL CLAUSE> IS NOT ARRAY OR STRUCTURE, AS IT MUST BE. 
RETURN

AT THIS POINT, WE FIND THE LOWEST MODE IN THE <ROW OF CLAUSE>, CONVERT
CIRCULAR LIST RESEMBLING A STRUCTURE, AND CALL THE LOWER SUBROUTINE.

CALL OUTDOA (7,7HSAVEU, LMHCRU, 10U+1, U, SAVEU, 10H, //)

CREATE A RUN-TIME ARRAY

STEP 1. GENERATE THE DESCRIPTOR.

ITEMP4=CONVERT (ITEMP3, 2R)
CALL OUTDOA (7, LMHCRU, 1H+, ITEMP4, 1H+, LMHCRU, 10H, //)
CALL OUTDOA (7, 10H010000000, 10H0000 + 1U1, 10H, //)
CALL OUTDOA (7, 10H0U+2, U, 10H10=17, U+1, 10H+U, //)
CALL OUTDOA (7, 10H000000100, 10H000000100, 10H+U, //)
CALL OUTDOA (7, CONVERT (ITEMP3, 2R), 10H*(UI183580H), //)

STEP 2. CALL LOWER AFTER GENERATING CIRCULAR LIST FOR LOWEST MODE.

ITEMP4=MINIMUM (N(I))
ITEMPS=IFETCH (I)
CALL SETIND (ITEMPS, ITEMP5, 1)
CALL SETIND (ITEMPS, ITEMP5, 2)
CALL LOWER (ITEMPS, ITEMP5, N(I), OPERAND(I), Z, P)

STEP 3. COMPLETE WORK ON THE RUN-TIME DATA ELEMENT.

ITEMP4=CONVERT (P, 2R T)
CALL OUTDOA (7, 10HPOP0, U, ITEMP4, 10H, P0P0, //)
I=I-1
OPERAND(I)=ITEMP3
RETURN

AT THIS POINT, A STRUCTURE IS GENERATED AT RUN TIME FROM THE ELEMENTS
<COLLATERAL CLAUSE>.

CALL OUTDOA (7, LMHCRU, 1H+, U, LMHCRU, 10H+1, U, SAVEU, 10H, //)
CALL OUTDOA (7, LMHCRU, 1H+, CONVERT (ITEMP3, 2R), 1H+, LMHCRU, 10H, //)
CALL LOWER (ITEMPS, LINK (ITEMP2, 2), OPERAND(I), Z, P)
GO TO 163

ENTRY PSI49A1

THE MEAN ASSOCIATED ON THE N-STACK WITH A <VOID CAST> IS 0. NOTE THAT
THE VALUE OF P DOES NOT CHANGE HERE BECAUSE OF THE ALGOL 68 CONVENTI
OF LEAVING THE DESIGNATION OF AN ASSIGNMENT AS ITS VALUE AND BECAUSE
OUR PSI33A2 RESETS P.

N(I)=0
OPERAND(I)=0
RETURN

ENTRY PSI48A2

P = P + 1
ITEMP=CONVERT (P, 2R T)
IF (.NOT.(ITEMP.EQ.OPERAND(I)))) GO TO 165
CALL OUTCSS (Z,OPERAND(I),1H,ITEMP,10H, //)
165 IF (SAMETF(N(I-1),N(I),REF)) GO TO 166
CALL ERROR (11OH THE A PRIORI MODE OF THIS <MODE CAST> DOES NOT MA
170H ITS A POSTERIORI MODE. )
RETURN
166 CALL OUTVAL (P,REF,OUTCODE,7,1000-Z)
I=I-1
OPERAND(I)=ITEMP
RETURN
ENTRY PSI49A1
THE N(I) MODE ASSOCIATED WITH A <VOID CAST> IS 0, AND WILL CAUSE AN E
MESSAGE FROM ANY CALL OF SAMETYP THAT TAKES A ZERO AS ONE OF ITS PARA
P=P-1
N(I)=0
OPERAND(I)=0
RETURN
ENTRY PSI49A2
DEREFERENCEING AND DEPROCURING MAY TAKE PLACE HEREF.
P=P+1
ITEMP=CONVERT(P,2R T)
IF (ITEMP.EQ.OPERAND(I)) GO TO 167
CALL OUTCSS (Z,OPERAND(I),1H,ITEMP,10H, //)
167 IF (SAMETF(N(I),N(I-1),REF)) GO TO 168
CALL ERROR (110HA PRIORI MODE OF THE <MODE CAST> DOES NOT MATCHIS
1 A POSTERIORI MODE. )
RETURN
168 CALL OUTVAL (P,REF,OUTCODE,2,1000-Z)
I=I-1
OPERAND(I)=ITEMP
RETURN
ENTRY PSI16A
THE INTERNAL REPRESENTATION OF REF AND PROC MODE PREFIXES IS CONSIST
BECAUSE OF LINKING FURTHER PREFIXES BY THE 2ND FIELD.
CALL PUTIND (I+IBYTE(CONT(N(I)),10),N(I),10)
ITEMP1=1
GO TO 171
ENTRY PSI16B
ITEMP=ITEMP1
171 ITEMP=IFETCH(I)
CALL SETIND (N(I),ITEMP,ITEMP1)
CALL SETIND (3RPRC,ITEMP,3)
N(I)=ITEMP
RETURN
ENTRY PSI15B1
K=K+1
CODELOC(K)=Z
RETURN
ENTRY PSI16C
ITEMP=IFETCH(1)
CALL SETIND (N(I),ITEMP,2)
CALL SETIND (N(I-1),ITEMP,1)
CALL PUTIND (IBYTE(CONT(N(I-1)),10),N(I),10)
I=I-1
N(I)=ITEMP
RETURN
ENTRY PSI77A1
IN THE PROCEDURE CALL, THE <PRIMA> MUST BE LOWERED TO ITS FIRST MOD
A PROCEDURE WITH PARAMETERS.
NI=N(I)
ITEMP=LOC(FITEMP)
IF (NI,GT,0) GO TO 173
172 CALL ERROR (110HPROCEDEUPE CALL ATTEMPTED ON A NON-PROCEDURE <PRIMA>)
RETURN
173 ITEMPE=CONT(NI)
ITEMP1=LINK(ITEMP,3)
THE MODE PREFIXES MUST BE SOME SEQUENCE OF REF,STOPPING AT THE FIRST
WITH PARAMETERS.
IF (ITEMP1,NE,3RREF) GO TO 175
ITEMP2=3RREF
174 ITEMP5=IFETCH(1)
CALL SETIND (ITEMP2,ITEMPS,3)
CALL SETIND (ITEMP5,ITEMP,2)
ITEMP3=ITEMP5
NI=LINK(ITEMP,2)
GO TO 173
175 IF (ITEMP1,NE,3RPDC) GO TO 172
IF (LINK(ITEMP,2,GT,0) GO TO 176
ITEMP2=3RPDC
GO TO 174
176 REF=ITEMP4
P=P+1
ITEMP4=CON(TEMP,2R T)
IF (OPERAND1,EQ,ITEMP4) GO TO 177
CALL OUTCD4 (2,OPERAND1,1H*,ITEMP4,16H, //)
OPERAND1=ITEMP4
TYPE1=6RSTORED
177 NI=NI
CALL OUTFAC (P,REF,OUTCODE,Z,1000-Z)

NEXT, STORE A LABEL ON N(I) FOR USE IN SETTING UP THE ACTUAL PARAMET
LIST, AND PROVIDE A JUMP AROUND THE ACTUAL PARAMS. LIST FOR LATER CONF

S=S+1

ITEM4=CONVERT(S,2R L)

N(I)=ITEM4

CALL OUTFAC (Z,0,0,ITEM4,10H1

TYPE(I)=7

CALL SETDIR (P+1,TYPE(I),2)

RETURN

ENTRY PSI78B1

THIS RULE CREATES A SEQUENCE OF LIST CELLS CONTAINING MODE INFORMATIO
RUN-TIME LABELS LOCATING WHERE EACH <UNITARY CLAUSE> BEGINS.

178 ITEMP=LINK(TYPE(I-1),2)

ITEMP3=50

ITEMP1=CONVERT(ITEMP,2R T)

IF (ITEMP1.EQ.,OPERAND(I)) GO TO 179

CALL OUTFAC (Z,OPERAND(I),1H*,ITEMP1,10H1

RETURN

179 P=ITEMP-1

CALL OUTFAC (Z,10HRETURN, Rj,10H(3847),,10HRJ,, //)

ITEMP2=IFETCH(2)

CALL STRING (N(I-1),ITEMP2+1)

STORES LABEL IN SECOND WORD.

CALL SETINC (N(I),ITEMP2,2)

STORES MODE LINK IN FIRST WORD.

?OPERAND(I)=ITEMP2

IF (ITEMP3.EQ.,0) RETURN

S=S+1

ITEMP2=CONVERT(S,2R L)

CALL OUTFAC (Z,ITEMP2,10H1

N(I)=ITEMP2

CALL SETDIR (ITEMP,TYPE(I),2)

RETURN

ENTRY PSI78A1

ABBREVIATED VERSION OF PSI78B1 WITHOUT INSERTION OF THE NEXT LABEL IN
OUTPUT CODE.

ITEMP3=0

GO TO 178

ENTRY PSI78B2

LINKS TOGETHER THE LIST OF LABELS AND MODE LINKS.
CALL SETINO (OPERAND(I), OPERAND(I-1), 1)
I=I-1
RETURN
ENTRY PSI77A2

OPERAND(I) CONTAINS THE LIST OF ENTRY LABELS AND DATA TYPES FOR EACH
ACTUAL PARAMETER. P WAS RESET TO ITS RETURN LEVEL - 1 BY PSI77A1. LH
SAVED BEFORE PARAMETER EVALUATION SO THAT ITS INITIAL VALUE BEFORE ST
PARAMETERS ON IT CAN BE RETRIEVED TO PASS TO THE PROCED, AT END OF
EVALUATION, CHECK FOR SAME LENGTH OF P-LISTS, TYPE(I-1) CONTAINS TH
TO INSERT A JUMP OVER THE PARAMETERS, AND N(I-2) CONTAINS THE A POSTE
PROCEDURE MODE.

ITEM=TYPE(I-1), AND .7777778
S=S+1
ITEM=CONVERT(S, 2R L)
OUTCODE(ITEM, 1)=ITEM1
OUTCODE(ITEM, 2)=10H,*
CALL OUTCOA2 (I, ITEM1, 10H1) /*
P=P+1
N=CONVERT(P, 2R T)
CALL OUTCOA4 (I, 7HSAVE U,, LHCurr, 3H+1#U,, 10HSAVE U, /*

LOWER<ACTUAL PARAMETERS> TO A POSTERIORI LEVEL, THEN STORE IN TURN IN
COUNT OF PARAMETERS
ITEM=OPERAND(I)
ITEM=LINK(INCONT(I(I-2)), 1)
ITEM1 AND ITEM3 ARE, RESPECTIVELY, LINKS TO THE A PRIORI AND A POSTE
PARAMETER LISTS.

180 ITEM2=CONT(ITEM1)
ITEM2=CONT(ITEM3)
ITEM2+ITEM5
IF (SAMEYPEL(INK(ITEM2, 2), LINK(ITEM4, 2), REFER)) GO TO 181
CALL ERROR(110H THE PARAMETER IN THE JTH POSITION OF THE PROCED
URE CALL DOES NOT MATCH ITS A POSTERIORI MODE, J IS...
CALL ERROR1(CONVERT(ITEM5, 2R )))
GO TO 180
181 CALL OUTCOA6 (I, 10HSAVE RETUR, 10HN, J+2#RETU, 3HRN, , CONT(ITEM1+1), 1
10H1#POP RETU, 10HN, /*
CALL OUTEVAL (P, REFER, OUTCODE, Z, 1000-2)
CALL OUTCOA6 (Z, LHCurr, 3H+1#LHCurr, 10H, /*
CALL OUTCOA6 (Z, ITEM1, 1H*, LHCurr, 10H, /*
ITEM1=LINK(ITEM2, 1)
ITEM3=LINK(ITEM4, 1)
IF ((ITEM1, GT, 0) AND (ITEM3, GT, 0)) GO TO 180
IF ITEM1, EQ, ITEM3) GO TO 162
CALL ERROR (110HILLEGAL PROCEDURE CALL--THE NUMBER OF ACTUAL AND F
NORMAL PARAMETERS DOES NOT MATCH.

P=P-1
I=I-2
RETURN

PASS PARAMETERS AT RUN TIME, AND TRANSFER CONTROL TO THE SUBROUTINE.

182 CALL OUTCDA4 (Z,8HP0P U,U,10HPARAMETERS,10H,POP U, //)
I=I-2
CALL OUTCDA4 (Z,OPERAND(I),8H(18-35),9HDEF LEVEL,6H, //)

BEFORE JUMPING TO THE SUBROUTINE, RAISE PROC LEVEL AND SAVE THE RUN-
TIME ENVIRONMENT.

CALL OUTCDA2 (Z,10HENVI0RNMENT,10HT SAVE, //)
CALL OUTCDA5 (Z,10HSAVE RETUR4N,2H4N,OPERAND(I),10H(IO+17)A JUMP,10HP,J
1+2=RETUR4N,10HRSN, //)
CALL OUTCDA5 (Z,8HJUMPL11,10HP0PRTURN,10HRSTORE EN,10HENVIRONMENT
17,104, //)
CALL OUTCDA3 (Z,8HVALUE + OPERAND(I),10H, //)
TYPE(I)=6RSTORED
P=P-1
RETURN

ENTRY PSI73A1

TAKES THE RUN-TIME PROCLEVEL AND THE LABEL AT THE BEGINNING OF <MOID
AND STORES IT IN A TP REGISTER, RESTORES DEFLEVEL ENVIRONMENT, JUMPS
ROUTINE COMPLETELY.

P=P+1
ITEM=CONVERT(P,2R T)
S=S+1
ITEM1=CONVERT(S,2R L)
CALL OUTCDA5 (Z,1H,ITEM1,1H,ITEM1,10H, //)
CALL OUTCDA5 (Z,10HPRO0 CLEVEL,1H,ITEM1,10H(18-35), //)
CALL OUTCDA6 (Z,2H,ITEM1,ITEM1,10MMASK ENVIRONMENT, 10H,, //)

THE MASK ENVIRONMENT ROUTINE TESTS DEFLEVEL. IF DEFLEVEL = 0, THEN N
RESTATE MEMENT. OTHERWISE THE ENVIRONMENT SAVE TABLE IS DUMPED ONTO THE
NAME TABLE AT THE LEVEL GIVEN BY DEF LEVEL.

I=I+1
OPERAND(I)=ITEM1
TYPE(I)=6RSTORED

W(I)=?7

W(I) SAVES THE LOCATION IN CODE FOR THE OVERJUMP

CALL PROGRAM(P,0)

ENTERS THE HASH TABLE AND DOES BLOCK BEGIN BOOKKEEPING

RETURN

.17495
ENTRY PSI73A2

ITEM=IFETCH(1)
CALL SETIND(3,RGC,ITEM,3)
CALL SETIND(N(I),ITEM,2)
CALL OUTCDA4 (Z,OPERAND(I),10H=VALUE,,13H(Return),,,10H
I/))
S=S+1
ITEM1=CONVERT(S,2R L)
I=I-1
ITEM2=N(I)
OUTCODEITEMP2,1ITEMP1
CALL OUTCDA2 (Z,ITEMP1,10H)//
N(I)=ITEM
CALL PRCEND(P,0)
RETURN

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

END
SUBROUTINE OUTEVAL (P,REF,OUTCODE,Z,MAX)
INTEGER P,REF,OUTCODE,Z,CONT,CONV
DIMENSION OUTCODE(MAX,6)
ITEM=CONVERT(P,2R T)
101ITEM3=LINK(CONT(REF),2)

C REF POINTS TO THE LIST CREATED BY SAMETP CONTAINING INFORMATION
C ABOUT THE CORRECT SEQUENCE FOR DEPROCEDURING AND DEREferENCING
C THE EXPRESSION CURRENTLY BEING EVALUATED.
C
IF (ITEM3.EQ.0) RETURN
IF (ITEM3.EQ.3) GO TO 102
C THE REMAINING ALTERNATIVE, WHICH MUST BE TRUE, IS THAT ITEM SIGNALS A
C PROCEDURE CALL. THE NUMBER OF PARAMETERS IS FOUND IN LINK 3 OF THE
C WORD POINTED TO BY REF.
C
ITEM4=CONVERT(LINK(CONT(REF),3),2R )
CALL OUTCDA4 (Z,ITEMP4,10H=NUMBER OF,10HPARAMETERS,8H,//)
P=P-LINK(CONT(REF),3)
ITEM=CONVERT(P,2R T)
CALL OUTCDA4 (Z,10HSAVE RETURN,3HN,,10HITEMP=REF(1,,10H30#47),//)

C CONTROL IS PASSED TO THE PROCEDURE9
C
CALL OUTCDA2 (Z,10HJ=2RETURN,10H, RJJ, //)
CALL OUTCDA5 (Z,10HPopRETURN,10HVALUE RETU,3HN*,ITEMP,10H,
1//)
CALL OUTCDA2 (Z,10HPop VALUE,10HRETURN, //)
REF=LINK(CONT(REF),1)
G) TO 101
C THE NUMBER OF REFERENCE LEVELS IS FOUND IN LINK 3 OF THE WORD POINTED
C TO G BY REF.
C
102 TEMP4=LINK(CONT(REF),3)
GO TO 104
103 TEMP4=ITEMP4-1
104 CALL OUTCDATA (Z,1H,ITEMP,2H,*,ITEMP,10H, //)
IF (ITEMP4.GT.0) GO TO 107
REF=LINK(CONT(REF),1)
GO TO 101
C
END
INTEGER FUNCTION MINIMUM(NI)
LOGICAL SAME
INTEGER REF,CONT
ITEMP=CONT(NI)
LINK1=LINK(ITEMP,1)
ITEMP=LINK(ITEMP,1)
101 ITEMP=CONT(ITEMP)
LINK2=LINK(ITEMP,1)
IF (SAMEYF(LINK1,LINK2,REF)) GO TO 102
C
AT 5, LINK2 IS THE LOWER DATA TYPE.
C
IF (SAMEYF(LINK2,LINK1,REF)) GO TO 103
CALL ERROR (62)
RETURN
102 LINK1=LINK2
103 ITEMP=LINK(ITEMP,1)
IF (ITEMP.GT.0) GO TO 101
MINIMUM=LINK1
RETURN
C
END
SUBROUTINE BALANCE (LINK1,ITEMP1,ITEMP2,S,P0,Z,OUTCODE)
LOGICAL SAME
INTEGER S,P,Z,OUTCODE,CONT,P0
ITEMP=LINK(CONT(ITEMP2),2)
101 S=S+1
ITEMP=CONVERT(S,2R L)
P=P+1
70 CALL STRING (ITEMP,ITEMP1+2)
CALL OUTCDATA (Z,ITEMP,10H, //)
CALL OUTCDATA (Z,10HSAVE RETUR,10HN,J+2=RETN,10RN, //)
CALL OUTCDATA (Z,CONT(ITEMP1+1),10H, //)
CALL OUTIVAL (P,REF,OUTCODE,Z,1000-Z)
CALL STRING (Z+1,ITEMP1+3)
C
HERE, THE ESCAPE ADDRESS IS PUT INTO THE THIRD LIST ELT. DATUM.
C SEE PS1B2D3 FOR DETAILS. NOTE THAT THE ROWS OF CLAUSE USE TICODELOCK
C AS THE VALUE RETURN ABOVE.
C
CALL OUTCDATA (Z,0,10H, //)
ITEMP=CONT(ITEMP1)
ITEMP2=LINK(CONT(ITEMP2),1)
IF (ITEMP1.GT.0) GO TO 101
IF (ITEMP2.EQ.0) RETURN
CALL ERROR1(THE ERROR IN COMPILER S BALANCE SUBROUTINE.
1
STOP
RETURN
END
SUBROUTINE LOOP82D (LINK, LABEL, OUTCODE)
DIMENSION OUTCODE (5000, 6)
INTEGER OUTCODE
ITEMP=INHALT (LINK+3)
101 OUTCODE (ITEMP, 1) = LABEL
ITEMP = LINK (CONT (ITEMP), 1)
IF (ITEMP, EQ, 0) RETURN
GO TO 101
END
SUBROUTINE LOWER (LENGTH, STANDARD, ROW, OPERANDI, Z, P0)
INTEGER ROW, STANDARD, OPERANDI, Z, P0, P, CONT, REF, CONVERT
LOGICAL SAFETY
LENGTH GIVES THE NUMBER OF FIELDS TO LOWER, STANDARD IS THE A POSTERIO
STRUCTURE, ROW IS THE A PRIORI MODE STRUCTURE, OPERANDI IS THE LIST OF
CLAUSE ENTRY LABELS.
ITEMP3 = CONVERT (P, 2R T)
IG = OPERANDI
I1 = STANDARD
I2 = ROW
P = P0
DO 102 J = 1, LENGTH
ITEMP1 = CONT (I1)
ITEMP2 = CCNT (I2)
LINK1 = LINK (ITEMP1, 2)
LINK2 = LINK (ITEMP2, 2)
IF (SAME (LINK1, LINK2, REF)) GO TO 101
CALL ERROR1 (THE FOLLOWING <ROW OF CLAUSE> FIELD DOES NOT
1 MATCH ITS A POSTERIORI MODE.
CALL ERROR1 (CONVET (J, 2R T)
1
FIND LABEL OF NEXT CLAUSE IN THE ROW, GENERATE A RETURN JUMP TO IT, T
CALL OUTEVAL TO LOWER IT.
BEGIN
CALL OUTDOA (Z, 1OHSAVE RETUR, 10HN, J+2+ RETU, 1OHRN, //)
CALL OUTDOA (Z, CONT (I+1), 10H, //)
CALL OUTEVAL (P, REF, OUTCODE, Z, 1OH9-Z)
CALL OUTDOA (Z, 6HU+1-U, ITEM3, 3OH-U, //)
101 I = CONT (10)
I1 = LINK (ITEMP, 1)
102 I2 = LINK (ITEMP, 2)
RETURN
INTEGER FUNCTION IMASH (NAME, MODE)
DIMENSION NAMES1 (197, 2), NAMES2 (197, 2), HASH1 (197, 2), HASH2 (197, 2)
ENTRY INITIAL

CALLED ON ENTRY TO THE COMPILATION OF A NEW PROGRAM.

PRLEVEL=0
BLOKNUM=0
ALCOUN=0
PRCOUN=0
MBCOUNT=0
HPCOUNT=0
L=IFETCH1(0)

INITIALIZES ALL COMPIL TIME LISTS

RETURN

ENTRY FINAL

CALLED WHEN COMPILER REQUESTS TALLY OF NAMES IN NAMES AND PROCS TABLE

NAME=MBCOUNT

MAXIMUM NAMES IN STATIC ENVIRONMENT

MODE=HPCOUNT

MAXIMUM NO. OF NAMES IN PROCEDURE ENVIRONMENT.

RETURN

ENTRY BLOKGN

BLOKNUM=BLOKNUM+1
STORAGE(BLOKNUM)=ALCOUN
CALL SETDIR(PROCNUM,STORAGE(BLOKNUM),2)
CALL SETDIR(NAME,STORAGE(BLOKNUM),3)

101 L=IFETCH2(0)

COMPILE-TIME LISTS BOOKKEEPING

RETURN

ENTRY BLOKEND

L=BLCOUNT-1
IF (L.LT.0) GO TO 103
OD 102 I=0,L
J=NAME1(BLCOUNT-I,1)
102 HASHI1(J,1)=0
103 BLCOUNT=STORAGE(BLOKNUM),AND,777778
PROCOUNT=LINK(STORAGE(BLOKNUM),2)
   .
NAME=LINK(STORAGE(BLOKNUM),3)
104 L=IFETCH(0)
   .
CALLS THE LIST GARBAGE COLLECTOR.
   .
RETURN
  ENTRY PROCBGN
   .
PRLEVEL=PRLEVEL+1
PSTORE(PRLEVEL)=PROCOUNT
   .
CALL SETDIR(NAME,PSTORE(PRLEVEL),2)
   .
GO TO 101
   .
  ENTRY PROCEND
   .
DELETES COMPILE TIME NAMES IN CURRENT PROCEDURE BLOCK, BUT LEAVES THE
   .
TIME LOCATIONS INTACT.
   .
   .
L=PROCOUNT-1-LINK(PSTORE(PRLEVEL),1)
   .
NAME=LINK(PSTORE(PRLEVEL),2)
   .
IF (L.LT.0) GO TO 106
   .
105 I=0,L
   .
J=NAMES2(PROCOUNT-I,1)
   .
106 PRLEVEL=PRLEVEL-1
   .
GO TO 104
   .
ENTRY IHASHA
   .
LOOK UP NAME AND MODE, OR LABEL AND LIST OF LOCATIONS TO BE FILLED, AND
   .
RUN-TIME ALIAS PRECEDED BY ~.
   .
LASTUSE=0
I=NAME
   .
IF (PRLEVEL.EQ.0) GO TO 111
   .
110 L=0,196,1
   .
   .
I=MOD(I+47*L,197)+1
   .
IA=HASH2(I,1)
   .
IF (IA.NE.0) GO TO 108
   .
IF (LASTUSE.GT.0) GO TO 107
   .
GO TO 111
   .
107 IHASHA=CONV(LASTUSE,2*P)
   .
MODE=LOCF(NAMES2(LASTUSE,2))
   .
RETURN
   .
108 IF (IA.NE.NAME) GO TO 109
   .
LASTUSE=HASH2(I,2)
   .
109 CONTINUE
   .
110 CONTINUE
   .
IF(LASTUSE.GT.0)GO TO 107
   .
NAME IS NOT WITHIN PROCEDURE ENVIRONMENT.
   .
111 00 115 L=0,196,1
I=MOD(I+47*L,197)+1
IA=HASH1(I,1)
IF (IA,NE.0) GO TO 113
IF (LASTUSE,GT.0) GO TO 112
CALL ERROR(110)THE TAG BELOW IS USED IMPROPERLY IT HAS NOT
BEEN DECLARED IN THIS OR ANY GLOBAL BLOCKS
) CALL ERROR1(NAME)
112 IHASH=CONVERT(LASTUSE,2R^V)
MODE=LOCF(NAMES1(LASTUSE,2))
RETURN
113 IF (IA,NE.NAME) GO TO 114
LASTUSE=HASH1(I,2)
114 CONTINUE
115 CONTINUE
IF (LASTUSE,GT.0) GO TO 112
 ENTRY IHASH
 ENTER NAME AND MODE ON NAMES TABLE, AND RETURN RUN-TIME ALIAS PREFIXED
116 I=NAME
 IF (LEVEL,EQ.0) GO TO 118
 DO 116 L=0,196,1
 I=MOD(I+47*L,197)+1
 IA=HASH2(I,1)
 IF (IA,NE.0) GO TO 117
 ENTRY THE NAME.
117 CONTINUE
 CALL ERROR (110)COMPILE-TIME NAME TABLE OVERFLOW DURING A PROCEDUR
1E DECLARATION.
) RETURN
118 PRCOUNT=PRCOUNT+1
 IF (PRCOUNT,GT.MCOUNT) MCOUNT=PRCOUNT
 HASH2(I,1)=NAME
 HASH2(I,2)=PRCOUNT
 NAMES2(PRCOUNT,1)=I
 NAMES2(PRCOUNT,2)=MODE
 MODE=LOCF (NAMES (PRCOUNT,2))
 FOR USE BY LABELS.
 IHASH=CONVERT(PRCOUNT,2R^P)
 RETURN
119 CONTINUE
 CALL ERROR (110)COMPILE-TIME NAME TABLE OVERFLOW OUTSIDE OF A PRO
1CEDUR DECLARATION.
) RETURN
120 BLCOUNT=BLCOUNT+1
 IF (BLCOUNT,GT.MBCOUNT) MBCOUNT=BLCOUNT
HASL{1(1;1)=NAME 1570
HASL{1(1;2)=BLCOUNT 1580
NAME{1(BLCOUNT,1)=I 1590
NAME{1(BLCOUNT,2)=MODE 1600
MODE=LOCF(NAME{1(BLCOUNT,2)) 1610
#SHAPE=CONVR(BLCOUNT,2)*V 1620
RETURN 1630

ENTRY HASHC 1640

THIS FUNCTION LOOKS UP OPERATORS IN THE NAME TABLE. WE ASSUME THAT AN 1650
OPERATOR NOT PRESENT ON THE TABLE IS EITHER INVALID OR WIRED IN TO THE 1660
RUN-TIME PILOT SYSTEM.

LASTUSE=0 1670
I=NAME 1680
IF (PLEVEL,0,0) GO TO 124 1690
d 127 L=0;196,1 1700
IA=MOD(I+47*L,197)+1 1710
IF (IA,NE,0) GO TO 121 1720
IF (LASTUSE,GT,0) GO TO 107 1730
GO TO 124 1740
121 T#=HASL{1(1,2) 1750

; COMPARE OPERAND MODES.
; I#={I,NAME,AND,NOT,SAMECVY(MODE,IB,REF))} GO TO 122 1760
LASTUSE=HASL{1(1,2) 1770
122 CONTINUE 1780
123 CONTINUE 1790
IF (LASTUSE,GT,0) GO TO 107 1800
d 126 L=0;196,1 1810
IA=MOD(I+47*L,197)+1 1820
IF (IA,NE,0) GO TO 125 1830
IF (LASTUSE,GT,0) GO TO 112 1840
IHASHE=0 1850
RETURN 1860
125 T#={HASH{1(1,2) 1870
I#={I,NAME,AND,NOT,SAMECVY(MODE,IB,REF))} GO TO 126 1880
LASTUSE=HASH{1(1,2) 1890
125 CONTINUE 1900

IF (LASTUSE,GT,0) GO TO 112 1910
RETURN 1920

END 1930