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 GP-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></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>
<tr>
<td>Table</td>
<td>Page</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>1</td>
<td>List Processing Primitives Used by the Compiler</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>1</td>
<td>Representation of mode declarations by compiler lists</td>
</tr>
<tr>
<td>1</td>
<td>Continued</td>
</tr>
<tr>
<td>2</td>
<td>Run-Time Representation of Array Descriptors</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 1 as a means of describing
the structure and translation of programs written in the ALGOL 68 language.

The context-free grammar of Appendix 1 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 1 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:

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

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

Rule (1) is exactly the sort of rule one would expect to find in a document such as the ALGOL 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:

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

\[
\text{MODE}/\psi_{25a}/<\text{mode indicant}> = /\psi_{25a2}/
\]

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

In this expansion of rule (1) above, we have added the notation "\(P_{25a}\)" 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\}, \text{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 "MODE" is recognized by the compiler, subroutine \(\psi_{25a1\} is called when the sequence of symbols

"\text{MODE}<\text{mode indicant}> = "

is recognized by the compiler, subroutine \(\psi_{25a2\} is called; and
the completion of the right-hand side of syntactic rule

_\Phi_{25a} \text{ causes subroutine } \phi_{25a3} \text{ 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 C.D.C. 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 $F$ = PROC(INT,$S$)REAL;
MODE $S$ = STRUCT(REF $S$ POINTER, $Q$ VALUE);
MODE $G$ = UNION($T$, $R$);
MODE $RS$ = REF $S$;
MODE $T$ = [1:5, 1:6]REF $F$;
```

**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 node 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>
</table>
| L@CF(X)    | address (X) + L@CF  
            | (The function returns the machine address  
            | of P0INTRAN variable X.)                                           |
| @@NT(I)    | [I] + @@NT                                                   |
| INHALT(I)  | [I] + INHALT                                                |
|            | (The value stored in the memory cell whose  
            | address is given by I is returned.)                                |
| BYTE(J,K)  | J(6*(K-1) + 6*K) + BYTE                                      |
|            | (The Kth character in word J is returned  
            | right justified with padded zeroes.)                              |
| LINK(J,K)  | J(16*(K-1) + 16*K) + LINK                                    |
|            | (The Kth address-sized field in word J is  
            | returned right justified with padded zeroes.)                     |
| PUTDIR(A,B,K) | A(0 + 5) + B(6*(K-1) + 6*K)                    |
|            | (The first character of A is put into the  
            | Kth character position of word B.)                                 |
| PUTIND(A,J,K) | 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.)                                   |
| SETDIR(A,B,K) | A(0 + 17) + B(18*(K-1) + 18*K)              |
| SETIND(A,J,K) | A(0 + 17) + [J](18*(K-1) + 18*K)       |

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

The purpose of IFETCH is to return the machine address of the first  
word in a list cell of K words stored sequentially in computer memory.  
The SAME(TYP 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 60, 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 "PRWC REF REF PRWC REAL", and J pointed to "PRWC REAL", the SAMETYP 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 "PRWC 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 SAMETYP function also incorporates algorithms for looking up the lists assigned to mode indicants appearing in I or J and for replacing these mode indicants 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 wherever 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:

\[
0 + \text{INDEX REGISTER } 1, \\
U - 1 \rightarrow U, \\
\text{LABEL 1: INDEX REGISTER } 1 + 1 \rightarrow \text{INDEX REGISTER } 1, \\
\text{INDEX REGISTER } 1 \rightarrow \text{SIZE : LABEL 2 ; ; } \\
\text{LACURR } 1 \rightarrow [U + \text{INDEX REGISTER } 1], \\
\text{LABEL 1 ,} \\
\text{LABEL 2 :} \\
\]

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 FORTRAN 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.

"CDELMC" is the system name for the second stack, a one-dimensional array with index K. As its name implies, the CDELMC stack saves information 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 CDELMC which enables us to produce reasonably efficient and non-redundant code in a single-pass compilation process. Of course, CDELMC 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 CDELMC stack, we see that the resulting compiler system potentially has the same computational ability as a Turing machine. This is because any one of the compiler subroutines could be written so as not to return control to the compiler, and could continue operation by manipulating the two stacks as 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 algorithm can in theory be written in a single-pass version for a translation 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 I 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:

\[
\begin{align*}
\langle \text{mode indicant} \rangle & \rightarrow \$<\text{name}>$
\langle \text{operator} \rangle & \rightarrow + | - | \ldots | + <\text{name}> + \\
\langle \text{call} \rangle & \rightarrow \langle \text{primary} \rangle \langle \text{actual parameters}\rangle \\
\langle \text{slice} \rangle & \rightarrow \langle \text{primary} \rangle \langle \text{indexes}\rangle \\
\langle \text{base} \rangle & \rightarrow 0 \# \text{T} \langle \text{label} \rangle
\end{align*}
\]

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

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

does not mean the same thing as

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

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 phrase terminates at the first labeled statement or the first statement followed by a jump. To translate such a declaration phrase properly, the compiler needs to know at the beginning of each statement in a declaration phrase 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 declaration prelude; rather, these interspersed statements are made to appear within the declarations in the blocks that legally may appear there.

The preceding problems of declaration preludes and uniquely identifiable 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:

\[
<\text{stowed declarer}> = \text{STRUCT} (\text{<fields>})
\]

\[\text{[<rows>] STRUCT (\text{<fields>})}\]

\[\text{[<rows>] <nonstowed declarer>}\]

\[\text{[<rows>] <mode indicant>}\]

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 com-
piler 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:

$P_{29a}$: <reference to actual declarer> + <actual mode declarer>/$^29a_1/$

$P_{2a}$: <nonstowed declarer> + REF/$^2a_1$/<mode declarer>/$^2a_2/$

$P_{30a}$: <reference to mode global generator> + HEAP/$^30a_1$/
      <actual mode declarer>/$^30a_2/$

In the second example, the rules

$P_{21a}$: <virtual mode declarer> + <mode declarer>/$^21a/$

$P_{22a}$: <actual mode declarer> + <mode declarer>/$^22a/$

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 $^21a$ and $^22a$ both
test the N-TYPE stack to see that all arrays declared within the
<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:

\[ P_{5a} : \langle \text{fields} \rangle + \langle \text{field} \rangle/\psi_{5a} / \]

\[ P_{5b} : \langle \text{fields} \rangle + \langle \text{field}, \langle \text{fields} \rangle/\psi_{5b} / \]

\[ P_{5a} : \langle \text{field}, \langle \text{field} \rangle, \langle \text{fields} \rangle/\psi_{5a} / \]

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 \( P_{5a} \), 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

\[ P_{7a} : \langle \text{rowers} \rangle + \langle \text{rower} \rangle/\psi_{7a} / \]

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

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 68 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 "LNCURR" is initialized to contain the characters "LOC", and is temporarily reset to contain the characters "HEAP" whenever compiler subroutines $30a1$ and $30a2$ in translation rule P 30a are called. Then, during the translation process, whenever the code for a storage reservation is put out, the contents of LNCURR (denoted in our notation below by "(LNCURR)") 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 $P_{13a}$ 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 $\phi_{4a1}$ records in the $\phi$DELSC stack where translated code for the descriptor begins, so that $\phi_{4a2}$ can erase any code that may have been produced while scanning the <array generator>.

For the recognized program sequence

\begin{verbatim}
[rivers]
\end{verbatim}

the compiler produces the following translated code:

```
SAVEU,
2 + (LACURR) + U,
(LACURR) + \textit{1 + no. of dimensions} \rightarrow \{LACURR\},
0 \rightarrow [U],
Translation of \text{<rivers>},
U = \textit{1 + no. of dimensions} \rightarrow \text{TEMP1},
0 \rightarrow [\text{TEMP1}],
[U](36 \rightarrow 53) \rightarrow \text{SIZE},
\text{SIZE} > 0: U \rightarrow [\text{TEMP1}](0 \rightarrow 17),
(LACURR) + \text{SIZE} \rightarrow \{LACURR\};
\text{no. of dimensions} \rightarrow [\text{TEMP1}](36 \rightarrow 53),
PPFU,
```

```
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 \( \psi_{452} \) 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:

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

Figure 2 below displays the storage structure used for run-time array
descriptors. The "\( i_1 u_1 \)" notation represents a two-bit code for indicating
whether or not the corresponding lower and upper bounds can be changed
dynamically. When \( i_1 (\text{or } u_1) \) 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>$l_1 u_1$</td>
<td>stride 1</td>
<td>upper bound 1</td>
<td>lower bound 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$l_n u_n$</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:

$P_{7a} : <\text{rows} > + \langle \text{rower}/\psi_{7a1}/ \rangle$

$P_{7b} : <\text{rows} > + \langle <\text{rows} >, \langle \text{rower}/\psi_{7b1}/ \rangle \rangle$

Subroutine $\psi_{7a1}$ is called after the translation of the first pair of bounds in an array declaration, and subroutine $\psi_{7b1}$ is called for each subsequent pair of bounds in the declaration. Primarily, $\psi_{7a1}$ and $\psi_{7b1}$ 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 $\psi_{7b1}$ gives an idea of this interaction:
\[ [U](18 + 35) - [U](0 + 17) \rightarrow \text{SIZE}, \]
(Subtracts the lower bound from the upper bound.)
\[ \text{SIZE} \times [U](36 + 53) + \text{SIZE}, \]
(Calculates the value of the next stride.)
\[ U + 1 + U, \]
\[ 0 + [U], \]
(Zeroes the next word in the descriptor.)
\[ \text{SIZE} > 0 : \text{SIZE} \rightarrow [U](36 \rightarrow 53); \]
(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 \( P_{3d} \):

\( P_{3b} \):  
<stowed declarer> + <array generator>/\( \psi_{3b1} \)/
\[ \text{STRUCT}(/\psi_{3b2}/ \text{<fields>})/\psi_{3b3} \]

\( P_{3d} \):  
<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 <mode indicant>.

This copy of the <mode indicant> is created by treating the <mode indicant> as a function whose value is the address of the next location of the storage reservoir specified by \( \text{LHcurr} \). 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 → INDEX REGISTER 2,
U - 1 + U,

LABEL 2:
INDEX REGISTER 2 + 1 → INDEX REGISTER 2,
INDEX REGISTER 2 > SIZE: LABEL 3:;
[LHCURR] + 1 + [U + INDEX REGISTER 2],
Translation of <mode indicant>,
(This includes a jump to the definition of the
<mode indicant> and return.)
LABEL 2.,

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 (LHCURR) was set
equal to the last address of this region by compiler subroutine $\psi_{aa2}$. 
Thus, the first address in any data structure created by the <mode
indicant> is given by "(LHCURR) + 1", and this address is the value used
to initialize the corresponding array elements iteratively.

Structure Declarations

In our implementation of ALOOL 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 $\psi_{aa}$ of Appendix 1, since it unes
essentially the same mechanisms as rule $P_{3b}$:

$$P_{3b}: \text{<stowed declarer>} \rightarrow \text{STRUCT}(/\psi_{3a1}/<\text{fields}>)/\psi_{3a2}/$$

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

```
SAVE U,
(LCURR) $\rightarrow$ U,
(LCURR) $+$ \{no. of fields\} $+$ (LCURR),
```

Translation of <fields>,

```
PUSH U,
```

The compiler subroutines used above in translating <fields> 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 <fields>. Thus, the code produced by $\psi_{3a1}$ and $\psi_{3a2}$ reserves space for the translated structure in LDC 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 $\psi_{5a1}$ or $\psi_{6a}$ as follows:

```
\{no. of intervening nonstowed fields\} $+$ U $\rightarrow$ U,
(LCURR) $+$ 1 $+$ \{U\},
```

Translation of the <field>,

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 LDC 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 <field>. 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} <\text{mode indiciant}> = <\text{actual mode declarer}>

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

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

In the above code, "LABEL 4" is the unique program label corresponding to the definition of the <mode indiciant>. "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{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-definition function, we need a function call in the form of a <mode indiciant> appearing in some declaration. This function call produces the following code:

\text{SAVE RETURN,}
\text{J + 2 \rightarrow RETURN,}
\text{LABEL 4:}
\text{POP RETURN}
For this calling sequence, LABEL 4 is used as the location of the particular <mode indcator> 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 "PSP 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 HRAP 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**

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

**The Compiled Code**

```plaintext
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](35 + 53),
 1 → @OPRAN REGISTER 2,
 1 + [U](55 + 55),
 @OPRAN REGISTER 2 + [U](0 + 17),
 5 + @OPRAN REGISTER 2,
 1 + [U](54 + 55),
 @OPRAN REGISTER 2 + [U](18 + 35),
 1 + [U](36 + 53),
 [U](14 + 35) - [U](0 + 17) → SIZE,
 SIZE + 1 + SIZE,
 U + 1 + U,
 0 → [U],
 SIZE > 0: SIZE + [U](36 + 53);
```
U - 2 + TEMP1,
0 + [TEMP1],
[u]([36 + 53] + SIZE,
SIZE > 0: U + [TEMP1](0 + 17),
LOC + SIZE + LOC;
1 + [TEMP1]([36 + 53],
PPOP U,
0 + INDEX REGISTER 3,
U-1 + U,
LABEL 8:
INDEX REGISTER 3 + 1 + INDEX REGISTER 3,
INDEX REGISTER 3 > SIZE: LABEL 9.;;
LOC + 1 + [U + INDEX REGISTER 3],
SAVE U,
LOC + U,
LOC + 2 + LOC,
1 + U + U,
LOC + 1 + [U],
SAVE RETURN,
J + 2 + RETURN,
LABEL (3$3$),
PPOP RETURN PPOP U,
LABEL 9.,
LABEL 9:
PPOP U,
[RETURN],
LABEL 10:
Bibliography

References Cited in Text


Appendix 1
A Partial Translation Grammar for ALGOL 68*

\[ P_{1a} : \text{<mode declarer> } \rightarrow \text{<stowed declarer>}/P_{1a} \]
\[ P_{1b} : \] (nonstowed declarer)/P_{1b} \\
\[ P_{1c} : \text{mode indicator}/P_{1c} \]
\[ P_{2a} : \text{<nonstowed declarer> } \rightarrow \text{REF}/P_{2a1} \text{ mode declarer }/P_{2a2} \]
\[ P_{2b} : \text{mode declarer }/P_{2b} \]
\[ P_{3a} : \text{<stowed declarer> } \rightarrow \text{STRUCT }/P_{3a1} \text{<fields>}/P_{3a2} \]
\[ P_{3b} : \text{<arraygenerator }/P_{3b1} \text{STRUCT}/P_{3b2} \text{<fields>}/P_{3b3} \]
\[ P_{3c} : \text{<arraygenerator }/P_{3c1} \text{<nonstowed declarer>}/P_{3c2} \]
\[ P_{3d} : \text{<arraygenerator }/P_{3d1} \text{<mode indicator>}/P_{3d2} \]
\[ P_{4a} : \text{<arraygenerator } \rightarrow [/P_{4a1} \text{<rowes> }]/P_{4a2} \]
\[ P_{5a} : \text{<fields> } \rightarrow \text{<field>}/P_{5a1} \]
\[ P_{5b} : \text{<field>}/P_{5b1} \text{<fields>}/P_{5b2} \]
\[ P_{6a} : \text{<field> } \rightarrow \text{<field mode declarer }/P_{6a1} \text{ field selector>}/P_{6a2} \]
\[ P_{7a} : \text{<rowes> } \rightarrow \text{<row>}/P_{7a1} \]
\[ P_{7b} : \text{<row>}/P_{7b1} \]
\[ P_{8a} : \text{<field mode declarer> } \rightarrow \text{<mode declarer> } \]
\[ P_{9a} : \text{<field selector> } \rightarrow \text{<name>}/P_{9a1} \]
\[ P_{10a} : \text{<row> } \rightarrow : /P_{10a1} \]
\[ P_{10b} : \] (lower bound:<upper bound)
\[ P_{11a} : \text{<lower bound> } \rightarrow \text{<bound>}/P_{11a1} \]
\[ P_{12a} : \text{<upper bound> } \rightarrow \text{<bound>}/P_{12a1} \]
\[ P_{13a} : \text{<bound> } \rightarrow \text{<formula>}/P_{13a1} \]
\[ P_{13b} : \text{EITHER}/P_{13b1} \]
\[ P_{13c} : \text{FLEX}/P_{13c1} \]

* This syntax has been tested, and is certified to be LR(1).
\[ P_{13} : \quad | \text{formula EITHER}/\psi_{13d}/ \]
\[ P_{13e} : \quad | \text{formula FLEX}/\psi_{13e}/ \]
\[ P_{14a} : \quad \langle\text{mode declarator} \rangle \rightarrow \langle\text{procdecl} \rangle \]
\[ P_{14b} : \quad | \langle\text{union mode decl} \rangle \]
\[ P_{14c} : \quad | \text{REAL}/\psi_{14c}/ \]
\[ P_{14d} : \quad | \text{INT}/\psi_{14d}/ \]
\[ P_{14e} : \quad | \text{BOOL}/\psi_{14f}/ \]
\[ P_{14f} : \quad | \text{CHAR}/\psi_{14f}/ \]
\[ P_{15a} : \quad \langle\text{procdecl} \rangle \rightarrow \text{PROC}/\psi_{15a}/ \]
\[ P_{15b} : \quad | \text{PROC}/\psi_{15b1}/\langle\text{virtual procplan} \rangle/\psi_{15b2}/ \]
\[ P_{16a} : \quad \langle\text{virtual procplan} \rangle + \langle\text{virtual parameters} \rangle/\psi_{16a}/ \]
\[ P_{16b} : \quad | \langle\text{virtual mode declarer} \rangle/\psi_{16b}/ \]
\[ P_{16c} : \quad | \langle\text{virtual parameters} \rangle/\langle\text{virtual mode declarer} \rangle/\psi_{16c}/ \]
\[ P_{17a} : \quad \langle\text{virtual parameters} \rangle + \langle\text{virtual mode declarer} \rangle/\psi_{17a}/ \]
\[ P_{17b} : \quad | \langle\text{virtual mode declarer} \rangle/\psi_{17b1}/\langle\text{virtual parameters} \rangle/\psi_{17b2}/ \]
\[ P_{18a} : \quad \langle\text{union mode decl} \rangle + \text{UNION } (\langle\text{nonunion indicants} \rangle)/\psi_{18a2}/ \]
\[ P_{19a} : \quad \langle\text{nonunion indicants} \rangle + \langle\text{mode indicant} \rangle/\psi_{19a}/ \]
\[ P_{19b} : \quad | \langle\text{mode indicant} \rangle, /\psi_{19b1}/\langle\text{nonunion indicants} \rangle/\psi_{19b2}/ \]
\[ P_{20a} : \quad \langle\text{mode indicant} \rangle \rightarrow \$ \langle\text{name} \rangle \$/\psi_{20a}/ \]
\[ P_{21a} : \quad \langle\text{virtual mode declarer} \rangle + \langle\text{mode declarer} \rangle/\psi_{21a}/ \]
\[ P_{22a} : \quad \langle\text{actual mode declarer} \rangle + \langle\text{mode declarer} \rangle/\psi_{22a}/ \]
\[ P_{23} : \quad \langle\text{formal mode declarer} \rangle + \langle\text{mode declarer} \rangle \]
\[ P_{24a} : \quad \langle\text{declaration} \rangle + \langle\text{mode declaration} \rangle \]
\[ P_{24b} : \quad | \langle\text{operator declaration} \rangle \]
\[ P_{24c} : \quad | \langle\text{initialization} \rangle \]
\[ P_{24d} : \quad | \langle\text{priority declaration} \rangle \]
$P_{25a}$: \textit{mode declaration} $\rightarrow$ M:\$/\text{mode indicant} = /\text{actual mode declarer}/$

$P_{26a}$: \textit{operator declaration} $\rightarrow$ $\emptyset$/\text{op.} = /\text{routine} /$

$P_{27a}$: \textit{initialization} $\rightarrow$ \textit{reference to actual declarer}$/\text{name list}$

$P_{27b}$: |\textit{reference to mode global generator}$/\text{name list}$

$P_{27c}$: |\textit{formal mode declarer} <\textit{tag} = /$\text{name list}/$\text{tertiary}$/

$P_{28a}$: <\textit{name list} > <\textit{tag}/$

$P_{28b}$: |<\textit{tag}: = /$\text{tertiary} /$

$P_{28c}$: |<\textit{name list}, <\textit{tag}/$

$P_{28d}$: |<\textit{name list}, <\textit{tag}: = /$\text{tertiary} /$

$P_{29a}$: <\textit{reference to actual declarer} $\rightarrow$ <\textit{actual mode declarer}$/$

$P_{30a}$: <\textit{reference to mode global generator} $\rightarrow$ \textit{HEAP}$/\text{actual mode declarer}$/

$P_{31a}$: \textit{program} $\rightarrow$ \$\text{ENTRY}$/\text{particular program}$/\text{EXIT}$/

$P_{32a}$: \textit{particular program} $\rightarrow$ <\textit{closed clause}$

$P_{33a}$: <\textit{closed clause} $\rightarrow$ \textit{BEGIN}$/\text{serial clause}$/\text{END}$/

$P_{33b}$: |(/$\text{serial clause})$/

$P_{33c}$: |(/$\text{row of clause})$/

$P_{34a}$: <\textit{serial clause} $\rightarrow$ <\textit{declaration prelude sequence};$<\textit{suite of clause train}$

$P_{34b}$: |<\textit{declaration prelude sequence};$<\textit{label completer}$<\textit{suite of clause train}$

$P_{34c}$: |<\textit{label completer}$<\textit{suite of clause train}$

$P_{34d}$: |<\textit{suite of clause train}$

$P_{35a}$: <\textit{declaration prelude} $\rightarrow$ <\textit{unitary clause};$<\textit{declaration prelude}$

$P_{35b}$: |<\textit{declaration}$

$P_{36a}$: <\textit{declaration prelude sequence};$ $\rightarrow$ <\textit{declaration prelude sequence};$

$P_{36b}$: <\textit{unitary clause};$ $\rightarrow$ <\textit{unitary clause}$

$P_{36c}$: <\textit{unitary clause EXIT} $\rightarrow$ <\textit{unitary clause}$\text{EXIT}$

$P_{36d}$: <\textit{label};$ $\rightarrow$ <\textit{label}$ ;$
\begin{align*}
P_{37a} & : \text{<suite of clause train> } \rightarrow \text{<unitary clause>}, \\
P_{37b} & : \text{|<labelstat>} \\
P_{37c} & : \text{|<labelstat><suite of clause train>} \\
P_{37d} & : \text{|<unitary clause;><suite of clause train>} \\
P_{38a} & : \text{|<labelstat> + <unitary clause;><label:>} \\
P_{38b} & : \text{|<unitary clause EXIT><label:>} \\
P_{38c} & : \text{|<labelstat><label:>} \\
P_{38d} & : \text{|<label completer> } \rightarrow \text{<label>} \\
P_{38e} & : \text{|<label completer><label:>} \\
P_{38f} & : \text{|<declaration prelude sequence> } \rightarrow \text{<declaration prelude> /}^{34b2/} \\
P_{38g} & : \text{|<declaration prelude sequence;><declaration prelude>} \\
P_{39a} & : \text{<label> } \rightarrow \text{<name>} \\
P_{40a} & : \text{<unitary clause> } \rightarrow \text{<tertiary>} \\
P_{40b} & : \text{|<confrontation>} \\
P_{40c} & : \text{|FOR <tag> FROM <serial clause> BY <serial clause> TO <serial clause> DO <unitary clause>} \\
P_{40d} & : \text{|FROM <serial clause> BY <serial clause> TO <serial clause> DO } \\
& \text{<unitary clause>} \\
P_{40e} & : \text{|WHILE <serial clause> DO <unitary clause>} \\
P_{41a} & : \text{<confrontation> } \rightarrow \text{<identity relation>} \\
P_{41b} & : \text{|<conformity relation>} \\
P_{41c} & : \text{|<reference to mode assignment>} \\
P_{41d} & : \text{|<mode cast>} \\
P_{42a} & : \text{|<reference to mode assignment> } \rightarrow \text{<reference to mode tertiary>} \\
& : = \text{<unitary clause>} \\
P_{43a} & : \text{|<reference to mode tertiary> } \rightarrow \text{<tertiary>} \\
P_{44a} & : \text{|<conformity relation> } \rightarrow \text{<reference to mode tertiary>} \\
& \text{|<conformity relator><tertiary>} \\
\end{align*}
\[ P_{45a} : \text{<conformity relator> } \rightarrow : : \]

\[ P_{45b} : \quad ::= \]

\[ P_{46a} : \text{<identity relations> } \rightarrow \text{<reference to mode tertiary> } \]

\[ <\text{identity relator}> \text{<reference to mode tertiary> } \]

\[ P_{47a} : \quad \text{<identity relator> } \rightarrow : = \]

\[ P_{47b} : \quad ::= \]

\[ P_{48a} : \quad \text{<mode cast> } \rightarrow \text{<virtual mode declare> } / \text{48a1/ } <\text{unitary clause> } / \text{48a2/} \]

\[ P_{49a} : \quad \text{<void cast> } \rightarrow : <\text{unitary clause> } / \text{49a/} \]

\[ P_{50a} : \quad \text{<tertiary> } \rightarrow \text{<formula> } \]

\[ P_{51a} : \quad \text{<formula> } \rightarrow <\text{formula<op><unary> } \]

\[ P_{51b} : \quad \text{<unary> } \]

\[ P_{52a} : \quad \text{<unary> } \rightarrow <\text{op><unary> } \]

\[ P_{52b} : \quad \text{<secondary> } \]

\[ P_{53a} : \quad \text{<priority declaration> } \rightarrow \text{PRIORITY<op><nonzero digit> } \]

\[ P_{54a} : \quad \text{<void cast> } \rightarrow <\text{mode cast> } \]

\[ P_{54b} : \quad \text{<void cast> } \]

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

\[ P_{61a} : \quad \text{<op> } \rightarrow + \]

\[ P_{61b} : \quad | - \]

\[ P_{61c} : \quad | * \]

\[ P_{61d} : \quad | / \]

\[ P_{61e} : \quad | / . / \]

\[ P_{61f} : \quad | + \]

\[ P_{61g} : \quad | - \]

\[ P_{61h} : \quad | \]

\[ P_{61i} : \quad | [ ] \]

\[ P_{61j} : \quad | = \]
\[ p_{61k} : \mid > \]
\[ p_{61l} : \mid < \]
\[ p_{61m} : \mid \geq \]
\[ p_{61n} : \mid \leq \]
\[ p_{61o} : \mid = \]
\[ p_{61p} : \mid \# \]
\[ p_{61q} : \mid ::= \]
\[ p_{61r} : \mid +::= \]
\[ p_{61s} : \mid *::= \]
\[ p_{61t} : \mid /::= \]
\[ p_{61u} : \mid ./::= \]
\[ p_{61v} : \mid \text{RE} \]
\[ p_{61w} : \mid \text{IM} \]
\[ p_{61x} : \mid \text{ODD} \]
\[ p_{61y} : \mid \text{SIGN} \]
\[ p_{61z} : \mid \text{ROUND} \]
\[ p_{61\alpha} : \mid <\text{name}> \]
\[ p_{62a} : \mid \text{<secondary> + <primary>} \]
\[ p_{62b} : \mid \{ \text{<cohesion>} \]
\[ p_{63a} : \mid \text{<cohesion> + <selection>} \]
\[ p_{63b} : \mid <\text{generator}> \]
\[ p_{64a} : \mid \text{<selection> + <name> \text{OF} <secondary>} \]
\[ p_{65a} : \mid <\text{generator}> = \text{<reference to mode global generator>} \]
\[ p_{65b} : \mid <\text{reference to mode local generator}> \]
\[ p_{66a} : \mid <\text{reference to mode local generator> + LOC <actual mode declarer}> \]
\[ p_{67a} : \mid <\text{primary> + <base}> \]
\[ p_{67b} : \mid <\text{closed clause}> \]
\[ p_{67c} : \mid <\text{conditional clause}> \]
P_68a:  <base> + <tag>
P_68b:  |<denotation>
P_68c:  |<slice>.
P_68d:  |<call>.
P_68e:  |@ <label>.
P_68f:  |SKIP.
P_68g:  |NIL.
P_69a:  <tag> + <name>.
P_70a:  <name> + <letter>.
P_70b:  |<name> <letter>.
P_70c:  |<name> <digit>.
P_70d:  |<name> *.
P_71a:  <denotation> + <number>.
P_71b:  |<character denotation>.
P_71c:  |<string denotation>.
P_71d:  |<bits>.
P_71e:  |<routine>.
P_72a:  <routine> → (*(<formal parameters>)<moid case>*)
P_72b:  |<moid cast pack>.
P_73a:  <moid cast pack> → (*/<_73a1/<moid cast>*)/<_73a2>.
P_74a:  <formal parameters> + <formal parameters>.
P_74b:  |<formal parameter> <formal parameter>.
P_75a:  <formal parameter> → <formal parameter>.
P_76a:  <formal parameter> + <formal mode declarer> <name>.
P_77a:  <call> → <primary>/*<_77a1/<actual parameters>*/<_77a2>.
P_78a:  <actual parameters> → <unitary clause>/*<_78a1>.
P_78b:  |<unitary clause>/*<_78b1/<actual parameters>*/<_78b2>.
P_79a:  <slice> → <primary> [indexer]
P 80a: <indexer> + <trimscript>
P 80b: |<indexer>, <trimscript>
P 81a: <trimscript> + <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/'$\_33a1'/<serial clause>
         THEN/'$\_82a2'/<serial clause>
         ELSE/'$\_82a3'/<serial clause>FI/'$\_82a4'/
P 82b: |IF/'$\_33a1'/<serial clause>THEN/'$\_82a2'/<serial clause>FI/'$\_82b3'/
P 82c: |CASE/'$\_33a1'/<serial clause>IN/'$\_82d2'/<row of clause>ELSEAC/'$\_82c3'/
P 82d: |CASE/'$\_33a1'/<serial clause>IN/'$\_82d2'/<row of clause>ELSE/'$\_82d3'/
         <serial clause>ESAC/'$\_82d4'/
P 83a: <row of clause> + <unitary clause>,<unitary clause>/'$\_83a'/
P 83b: |<unitary clause>,<row of clause>/'$\_83b'/
P 84a: <unitary clause>,<row of clause>/'$\_84a'/
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>
\[
P_{90a}: \quad \text{octal integer} \rightarrow \text{octal digit}
\]
\[
P_{90b}: \quad |\text{octal integer} \rightarrow \text{octal digit}|
\]
\[
P_{91a}: \quad \text{octal digit} \rightarrow 0
\]
\[
P_{91b}: \quad |\text{nonzero octal}|
\]
\[
P_{92a}: \quad \text{nonzero octal} \rightarrow 1
\]
\[
P_{92b}: \quad \text{nonzero octal} \rightarrow 1
\]
\[
P_{92c}: \quad \text{nonzero octal} \rightarrow 1
\]
\[
P_{92d}: \quad \text{nonzero octal} \rightarrow 1
\]
\[
P_{92e}: \quad \text{nonzero octal} \rightarrow 1
\]
\[
P_{92f}: \quad \text{nonzero octal} \rightarrow 1
\]
\[
P_{92g}: \quad \text{7}
\]
\[
P_{93a}: \quad \text{digit} \rightarrow 0
\]
\[
P_{93b}: \quad |\text{nonzero digit}|
\]
Appendix II: TRANSLATION RULES WRITTEN IN FORTRAN

SUBROUTINE CLEAN
ENTRY PSI1A
ENTRY PSI1C
ENTRY PSI1E
ENTRY PSI1A1
ENTRY PSI1A2
ENTRY PSI2A
ENTRY PSI2A1
ENTRY PSI2A2
ENTRY PSI2B
ENTRY PSI3A1
ENTRY PSI3A2
ENTRY PSI3B1
ENTRY PSI3B2

C CODE FOR THIS NONSTOWED DECLARER IS DESTROYED AT THIS POINT. INFORMA
C ABJIT FORMAL DECLARERS IS PASSED ON.

C IF (LINK(CONN(I),1),ED,3RF)M GO TO 101
ITEMP=3RFCT
GO TO 102
101 ITEM=IFETCH(1)
102 ITEM=IFETCH(1)
CALL STRIND (4LORF,ITEMP)
CALL SETIND (N(I),ITEMP,2)
CALL SETIND (ITEMP,ITEMP,1)
N(I)=ITEMP
Z*CODELOC(K)
CALL REDUCE (K)
RETURN

C ENTRY PSI2B
C CALL SETIND (3RFCT,N(I),1)
RETURN
C ENTRY PSI3A1
C ENTRY PSI3A2
C ENTRY PSI3B1
C ENTRY PSI3B2
C
C A STORAGE RESERVATION COMMAND FOR THE STRUCTURE IS INSERTED HERE
C BY SUBROUTINE PSI3A2

K=K+1
CODELOC(K)=Z+3
K=K+1
CODELOC(K)=Z+4
FIELDNO(K)=1
CALL OUTCDA2 (Z,6HSAVEU,10H //)
CALL OUTCDA2 (Z,LHCCURR,10H + U, //)
CODE FOR THE OUTER STRUCTURE AND ITS FIRST FIELD IS INSERTED HERE BY
RULE PSI3A2.
7=7+i
CALL OUTCDA2 (Z,LHCCURR,10H + U1 //)
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(I)
CALL STRING (4LIST,ITEM)
CALL SETINC (N(I),ITEMP,2)
CALL SETINC (LINK(CONT(I)+1),ITEMP,1)
COUNTNO=ITEME(CONT(N(I)),10)
CALL PUTINC (COUNTNO,ITEMP,10)
N(I)=ITEMP
CALL REDUCE (K)
CLEAR THE STACK OF INFORMATION REGARDING THE FIELDS.
INDEX=CODELOC(K)
CALL OUTCDA3 (Z,LHCCURR,2H+,CONVERT(COUNTNO,2R),2H+,LHCCURR,7H,
1 //)
IF (COUNTNO,GT,FIELDNO(K)) GO TO 103

C IF EQUAL, THIS MEANS THAT NONE OF THE STRUCTURE FIELDS ARE STOWED,
C AND HENCE, NO SUBSTRUCTURES WERE CREATED.
C
ITEM=CODELOC(K)-2
CALL OUTCDA1 (ITEMP,2H//)
CALL OUTCDA1 (ITEMP,2H//)
GO TO 104
103 CALL OUTCDA1 (7,10HPOPU, //)
104 CALL REDUCE (K)
RETURN

ENTRY PSI3B1
ENTRY PSI3C1
C
ITEM=CONVERT(R,2RIR)
R=R+1
IF (R,GT,MAXR) MAXR=R
CALL OUTCDA3 (Z,3H+,ITEMP,10H, //)
CALL OUTCDA2 (Z,10H-1 + U,10H //)
S=S+1
K=K+1
CODELOC(K)=CONVERT(S+1,2R L)
K=K+1
CODELOC(K)=CONVERT(S,2R L)
CALL OUTCD2 (Z, CODELOC(K), 10H  )  //
CALL OUTCD4 (Z,ITEMP,SH + 1*,ITEMP,10H,  //
S=S+1
CALL OUTCD4 (Z,ITEMP,SH > SIZE 1, CODELOC(K+1),10H, 1;  //
CALL OUTCD4 (Z,LHCUW* 10H 1 = (U + 1ITEMP,10H),  //
RETURN

ENTRY PSI323

THISROUTINE IS ALMOST PSI3A2 FOLLOWED BY PSI3D2, WITH FACILITIES FOR
CARRYING OVER INFORMATION ABOUT VIRTUAL OR ACTUAL DECLARATIONS.

ITEMP=IFETCH(1)
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,LHCUW,SH +,CONVERT(COUNTNO,2R ),5H  )
1HCURR,10H,  //
IF (COUNTNO,GT,FIELDNO(K)) GO TO 105
INDEX=CODELOC(K)*2
CALL OUTCD1(INDEX,2H/)
CALL OUTCD2 (INDEX,2H/)
GO TO 106
105 CALL OUTCD1 (Z,10HPOP,  //
106 CALL REDUCE (K)
CALL OUTCD2 (Z, CODELOC(K), 10H,  //
X=X-1
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.

ITEMP=LINK(CONT(N(I)),1)
ITEMP=LINKCAT(N(I-1),1)
IF (ITEMP,=3RACT,AND,ITEMP,=,EQ,3RACT) GO TO 108
IF (ITEMP,=3FRHM,OR,ITEMP,=,EQ,3FRHM) GO TO 107
CALL SETIND (3FRHM,N(I-1),1)
GO TO 108
107 CALL SETIND (3FRHM,N(I-1),1)
108 I=I-1
R=R-1
RETURN

ENTRY PSI3C

AT THIS POINT, ARRANGEMENTS ARE MADE FOR SUBSCRIBING OF BIT AND
C CHARACTER ARRAYS. THERE IS NO NEED TO ERASE ANY CODE FOR NONSTOWED
DECLARERS, SINCE RULE'S Psi2A2, Psi14, Psi18, AND PSI19 DO THIS.

        ITEM=LINK(CONT(I),31)
        IF (ITEM.EQ.3BIT) GO TO 110
        IF (ITEM.EQ.3RAYT) GO TO 112
109 CALL SetIND (N(I),N(I-1),2)
        I=I-1
        RETURN
110 INC=10H/60 * (U -
111 R=R+1
        S=S+1
        IF (R.GT.MAXR) MAXR=R
        ITEM=CONVERT(R,2RIR)
        ITEM=CONVERT(S,2R L)
        CALL OUTCOA3 (Z,3H6* ,ITEM,10H, /*
        CALL OUTCOA6 (Z,ITEM1,6H )TU-,ITEM,9H1(56=53),INC,ITEM,10H1(3
        16=53),10H /*
        CALL OUTCOA4 (Z,ITEM,6H +1 +ITEM,10H, /*
        CALL OUTCOA5 (Z,ITEM,10H=DIMENSION,1H4,ITEM1,10H++,11 /*
        R=R-1
        GO TO 109
112 INC=10H12* [U -
        GO TO 111
        C
        ENTRY PSI3C2
        C
        CALL OUTCOA2 (Z,CODELOC(K),10H, /*
        K*K+1
        CALL OUTCOA (Z,CODELOC(K),10H1 /*
        CALL REDUCE (K)
        CALL SETIND (N(I),N(I-1),2)
        I=I+1
        R=R-1
        RETURN
        C
        ENTRY PSI4A1
        C
        K*K+1
        CODELOC(K)=Z
        FIELDNO(K)=0
        CALL OUTCOA1 (Z,10H0SAVEU, /*
        CALL OUTCOA2 (Z,LMCURR,10H+2 +U /*
        Z=Z+1
        CALL OUTCOA2 (Z,9H00 = [U1,,10H /*
        C
        THIS OUTPUT CODE MAKES U THE INDEX OF THE DESCRIPTOR BEING CREATED.
        A SPACE IS LEFT IN THE OUTPUT CODE FOR LATER INSERTION OF A STATEMENT
        OF THE FORM LMCURR+DIMENSION=LMCURR THAT RESERVES SPACE FOR THE
        ARRAY DESCRIPTOR ON EITHER THE HEAP OR THE LOO STACK. RULE PSI4A2
        MAKES THIS INSERTION
        C
        RETURN
        C
        ENTRY PSI4A2
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 PUTING (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 (3VRT,N(I),1)

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

114 Z=CODELOC(K)
K=K+1
RETURN
115 CALL SETIND (3ACT,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 PSI781 STORED THE SIZE OF THE ARRAY
IN U AT RUN TIME, WHERE U CURRENTLY THE ADDRESS OF THE WORD THAT
FOLLOWS THE DESCRIPTOR IN MEMORY.

CALL OUTCD6 (ITEM,LHcurr,4H,+ ITEM1,4H +LHcurr,5H, )
CALL REDUCE (K)
CALL OUTCD4 (Z,3HU-,ITEM1,8H*TMP1,10H, )
CALL OUTCD2 (Z,10H600*(ITEM1),10H, )
CALL OUTCD2 (Z,7HSIZE+Z,10H6U*(ITEM1),10H, )
CALL OUTCD3 (Z,LHCurr,10H+SIZE +LHCurr,10H11)
CALL OUTCD3 (Z,ITEM1,10H-1*(ITEM1),10H35,53, )

AT THIS POINT, U IS THE ADDRESS OF THE FIRST ARRAY ELEMENT.

RETURN

ENTRY PSI5A1
ENTRY PSI6A

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 INCREASES A GENERAL COUNT OF
FIELDS. IF STOWED OR MODE INDICANT, PUTS OUT CODE FOR INCREMENTS RUN
TIME ALLOCATION POINTER TO GENERATE A CONNECTION BETWEEN A FIELD AND
ITS DATA STRUCTURE.

IF (TYPE(I),EQ.6)STOWED) GO TO 114
    Z=CODELOC(K)+1
    FIELDDO(K)=1+FIELDNO(K)
    GO TO 117
114 ITEM=CODELOC(K)
    CALL OUTCDP2 (ITEM,CONVERT(FIELDNO(K),2R),I3H+U*,1/)
    CALL OUTCDP2 (ITEM,LHCUR,10H*U,1/)
    FIELDNO(K)=1
117 CODELOC(K)=Z+1
    Z=Z+2
    RETURN

ENTRY PS1501

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),1)
CALL PUTIND (1+1BYTENC(N(I)),1),N(I-1),1)
ITEM=LINK(CONT(N(I-1)),N(I),1)
ITEM1=LINK(CONT(N(I)),1)
IF (ITEM,EO,3RACT.AND.,ITEM1,EO,3RACT) GO TO 120
IF (ITEM,EO,3RACT.OR.,ITEM1,EO,3RACT) GO TO 116
IF (ITEM,EO,3RACT.OR.,ITEM1,EO,3RACT) GO TO 119
CALL SETIND (3RVT,N(I-1),1)
    GO TO 120
118 CALL SETIND (3RFRM,N(I-1),1)
    GO TO 120
119 CALL SETIND (3RVT,N(I-1),1)
120 I=I+1
    RETURN

ENTRY PS151

CALL SETDIR (1,FIELDNO(K),1)
CALL OUTCD2 (Z,10H+U(1)),10H+36*,10H53)
CALL OUTCD2 (Z,10H+U(1)),10H+35*,10H+6U*1*)
    RETURN

ENTRY PS17B1

COUNTS UP THE NUMBER OF DIMENSIONS, STORES THE STRIDE IN THE CURRENT
DESCRIPTOR WORD, AND ZEROS THE NEXT DESCRIPTOR WORD.

CALL SETDIR (1+LINK(FIELDNO(K)),1),FIELDNO(K),1)
CALL OUTCDAS (Z,10H+U(18*35),10H+U(0*17),10H+SIZE,1/)

CALL CALCULATES UPPERBOUND-LOWERBOUND.
CALL OUTCDA3 (Z, 10HSIZE * (U1(3, 1046=53) * SIZE, 10H), //)
CALL OUTCDA1 (Z, 10HU+1=U, //)
7=I+1
CALL OUTCDA4 (Z, 6H0+U), 7HSIZE=01, 10HSIZE=U1(3, 1046=53) //)
RETURN
ENTRY PS18A

GENERATES A CELL IN THE COMPIL-Time LIST COPY OF A STRUCTURE
DECLARATION.
ITEMP=IFIETCH(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 PS19A
I=I+1
N(I)=NAME
RETURN
ENTRY PS110A

THIS ROUTINE COUNTS VIRTUAL BOUNDS IN AN ARRAY DECLARATION.
CALL SETDIR (1+LINK(FIELDNO(K),2),FIELDNO(Y),2)
NOTE THAT ANYTHING BUT ZERO IN THE SECOND FIELD OF FIELDNO INDICATES
THAT THE ARRAY IS NOT AN ACTUAL ARRAY.
RETURN
ENTRY PS111A
ITEMP1=10H0U1(55=55)
ITEMP2=10H0U1(0+17)
GO TO 123
ENTRY PS112A
ITEMP1=10H0U1(54=54)
ITEMP2=10H0U1(18=35)
123 GO TO (124, 126, 128, 125, 125), BOUND
124 CALL OUTCDA3 (Z, 5H 01*, ITEMPL=10H, //)
125 CALL OUTCDA4 (Z, CONVERT(P,IZ T),14*,ITEMP1,18H, //)
P=P+1
I=I-1
125 RETURN
ENTRY PSI12B
SOUND=2
RETURN
ENTRY PSI13C
SOUND=3
RETURN
ENTRY PSI13A
SOUND=1
GO TO 127
ENTRY PSI12D
SOUND=4
GO TO 127
ENTRY PSI13E
SOUND=5
127 IF (SAMEYF(N(I),INTEGER,REF)) GO TO 128
CALL ERROR(INVALIDNON-INTEGER ARRAY BOUND IN ROUNOSLIST.
1 RETURN
128 CALL OUTIVAL (P,REF,OUTCODE,Z,1000-Z)
RETURN
ENTRY PSI14C
ITEMP=3RREA
GO TO 130
ENTRY PSI14D
ITEMP=3RINT
GO TO 130
ENTRY PSI14E
ITEMP=3RRAI
GO TO 130
ENTRY PSI14F
ITEMP=3RBYT
GO TO 130
ENTRY PSI15A
129 ITEM=30PRC
130 I=I+1
  ITEMP2=IFETCH(1)
  CALL SETIND (ITEMP,ITEMP2,3)
  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 (I+IBYTE(CONT(N(I)),10),N(I),10)
ITEMP1=2
GO TO 131
ENTRY PSI16B
ITEMP1=1
131 ITEMP=IFETCH(1)
  CALL SETIND (N(I),ITEMP,ITEMP1)
  I=I-1
  CALL SETINC (ITEMP,N(I),2)
  RETURN
ENTRY PSI16C
ITEMP=IFETCH(1)
CALL SETINC (N(I),ITEMP,1)
CALL SETIND (N(I-1),ITEMP,2)
CALL PUTINC (IBYTE(CONT(N(I-1)),10),N(I),17)
I=I-2
N(I)=ITEMP
RETURN
ENTRY PSI17A
ITEMP1=1
GO TO 132
ENTRY PSI17B1
ITEMP1=0
132 ITEMP=IFETCH(1)
  CALL SETIND (3PRM,ITEMP,3)
  CALL SETINC (N(I),ITEMP,2)
CALL PUTINC (ITEM1,ITEM,10)
N(II)=ITEM
RETURN

ENTRY PSI1782

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

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

ENTRY PSI18A1

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

ENTRY PSI18A2

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

ENTRY PSI19A

ENTRY PSI19B1

ITEM=IFETCH(1)
CALL SETIND (3RUNI,ITEM,3)
CALL SETIND (N(II),ITEM,2)
N(II)=ITEM
RETURN

ENTRY PSI19B2

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

ENTRY PSI20A

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

THE ALIAS FUNCTION LOOKS UP THE RUN-TIME PROGRAM LABEL ASSIGNED TO THE MODE INDICATOR, AND ADDS THE INDICATOR TO THE COMPILER DECLARATION TABLE IF NECESSARY.
CALL OUTCOA (7, 10HSAVE RETUR, 2H, 10HJ + 2#RETURN, 3H, */)
CALL OUTCOA (7, NAME, 2H, 10HPOP RETURN, 10H, */
RETURN
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 = LINK(CONT(I), 1)
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 = LINK(CONT(I), 1)
IF (ITEM .EQ. 3RACT .OR. ITEM .EQ. 3RVCT) 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 OUTCOE (7, NAME, 10H1 */
C THE MODE INDICANT WAS DECLARED BY PSI20A, AND NAME CONTAINS THE RUN-
TIME LABEL ASSOCIATED WITH THE INDICANT.
FIELDNO(K) = MODE
C MODE IS THE INDEX OF THE COMPILER MODES TABLE.
RETURN
ENTRY PSI25A3

CALL OUTCODA3 (Z,9HRETURN=RJ,8H(30=47),10HRJ,, //)
S=S+1
ITEM1=CONVERT(S,2R L)
ITEMP=CODELOC(K)+1
CALL OUTCOD2 (ITEMP,ITEMP1,10H,, //)
CALL OUTCODA2 (Z,ITEMP1,10H1,, //)
ITEMP=FIELDNO(K)
MDES(ITEMP,3)=N(I)

ASSIGNS COMPILER DATA STRUCTURE TO MODE INDICANT.

I=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 OUTCODP2 (Z,NAMe,10H1,, //
FIELDNO(K)=OP
RETURN

ENTRY PSI26A3

CALL OUTCODA3 (Z,9HRETURN=RJ,8H(30=47),,10HRJ,, //)
S=S+1
ITEMP1=CONVERT(S,2R L)
ITEMP=CODELOC(K)+1
CALL OUTCODP2 (ITEMP,ITEMP1,10H,, //)
ITEMP=FIELDNO(K)
OPS(ITEMP,3)=N(I)
CALL OUTCODA2 (Z,ITEMP1,10H1,, //)
I=I-1
CALL REDUCE (K)
RETURN

ENTRY PSI27A

THIS ROUTINE ACMS AN EXTRA LEVEL OF REFERENCING TO THE DECLARER, BOTH I
COMPILER REPRESENTATION AND THE RUN-TIME CODE9 LHCURR CARRIES INFORMATION
ABOUT WHETHER LOC OR HEAP IS IN USE.

S=S+1
ITEMP=CONVERT(S,2R L)
FIELDNO(K)=ITEMP
ITEMP1=CODELOC(K)+1
CALL OUTCQP6 (ITEMP1,ITEMP4,4H,1+,LHCURR,1H,*,LHCURR,0H,   //)
ITEMP1=ITEMP1+2
S=S+4
ITEMP=CONVERT(S,2R L)
133 CALL OUTCQP2 (ITEMP1,ITEMP,10H,  //)
CALL OUTCDAS (Z,10HRETURN+ RJ,10H(30*47), ,10HRJ,   //)
CALL OUTCDAS (Z,ITEMP,10H1 //)
ITEMP1=IFETCH(1)
CALL SETIND (3REF,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 PSI2DA1.
S=S+1
ITEMP1=ITEMP
FIELDNO(K)=ITEMP
ITEMP1=CODELOC(K)+1
CALL OUTCQP2 (ITEMP1,ITEMP,10H1 //)
S=S+1
ITEMP1=ITEMP1+1
ITEMP=CONVERT(S,2R L)
G0 TO 133
ENTRY PSI27C1
THE TAG RULE PSI69A HAS THE EFFECT OF ENTERING THE NAME ONTO THE NAMES
TABLE, AND LEAVES THE RUN-TIME ALIAS OF <TAG> IN THE VARIABLE NAME.
CODING FOR THE VIRTUAL DECLARER IS DESTROYED
OPERAND(I)=NAME
TYPE(I)=9RNONSTORED
Z=CODELOC(K)
RETURN
ENTRY PSI27C2
134 IF (SAMEYPFN(I-1,N(I),REF),AND,REF,EQ,0) GO TO 134
CALL ERROR(10HATTEMPT TO INITIALIZE A DECLARED VARIABLE
USING A <TERTIARY> OF A DIFFERENT MODE.
RETURN
134 P=P+1
ITEMP=CONVERT(P,2R T)
CALL OUTCDAS(Z,OPERAND(I),1H,*,ITEMP,10H,  //)
OPERAND(I)=ITEMP
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(Z, 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 (Z, NCURR, 4H, 1N, NAME, 10H) //
CALL OUTCDA3 (Z, 10H, RUSER, 10H, J+2, RRET, 10HR, //)
CALL OUTCDA2 (Z, FIELDNO(K), 10H, //)
CALL OUTCDA2 (Z, 10HP, RETURN, 10H, //)
ENTRY PSI28B1
ENTRY PSI28D1

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

ENTRY PSI28B2
ENTRY PSI28D2

THE DATA TYPES OF <TERTIARY> AND <TAG> ARE COMPARED. THE <TERTIARY> I
IS REFERENCED AND DEPROCEDURED TO ONE LEVEL OF REFERENCE BELOW N(I-1).
NOTE THAT THE FIRST PARAMETER OF SAMETYP IS THE ONE TO BE LOWERED.

IF (SAMETYP(NI), LINKCONT(N(I-1), 1), REF)) GO TO 169
CALL PRDR (110, THE <TERTIARY> ASSIGNED TO THE <TAG> OF THIS MODE
DECLARATION DO NOT POSSESS THE SAME MODE AS ITS ASSIGNEE.)
RETURN

169 IF (TYPE(I), NE, 9RNONSTORED) GO TO 170
P = P+1
ITEM = CONVRT(P, 28, T)
CALL OUTCDA4 (Z, OPERAND(I), 1H, ITEM, 10H, //)
170 CALL OUTDEVAL (P, REF, OUTCODE, 7, 1000-Z)
I = I+1
CALL OUTCDA2 (Z, ITEM, 1H, OPERAND(I), 10H, //)
RETURN
ENTRY PSI29A1
ENTRY PSI30A1

ITEM = IFETCH(1)
CALL SETINS (3RREF, INTEMP, 3)
CALL SETINS (N(I), INTEMP, 2)
N(I) = INTEMP
RETURN
ENTRY PSI30A2
LHECURR=3HLOC
CALL OUTCOA3 (Z,10H=GENERAT,10HOR VALUE, 10HPOP U, //
ENTRY PSI31A1
CALL INITIAL(NAME,MODE)
ENTRY PSI31A2
CALL FINAL(NAME,MODE)
ENTRY PSI33A1
ENTRY PSI33B1
LHECURR=3HLOC
CALL BLOK9GN(P,MODE)
ITEMP1=Z+3
K=K+1
CODELOC(K)=ITEMP1
K=K+1
CODELOC(K)=ITEMP1
RESERVES SPACE FOR THE TRANSLATION OF INITIALIZATION DECLARATIONS AND
FOR MANAGING RUN-TIME STORAGE ALLOCATION.
Z=Z+4
ENTRY PSI33A2
ENTRY PSI33B2
CALL BLOKEND(P,MODE)
ENTRY PSI33A2
ENTRY PSI33B2
RETURN
ENTRY PSI33A2
ENTRY PSI33B2
CALL BLOKEND(P,MODE)
ENTRY PSI33A2
ENTRY PSI33B2
RETURN
ENTRY PSI34A

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

ITEM=CODELOC(K)-2
ITEM1=ITEM-3
I=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 PSI33B1.

ITEM=CODELOC(K)-2
"CALL OUTC0P2 (ITEM,10H1+RLOKNUM,10HALOKNUM, //)
CALL OUTC0P3 (ITEM,4HLOC,10HSTORAGEBL,12HOKNUM), //)
CALL OUTC0A6 (Z,10HSTORAGEBL,12HOKNUM,LOC,1H,10HALOKNUM-1,10H\n 10KNUM,.10H //)
RETURN

ENTRY PSI36A

C.EARS 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/PSI82D2/<ROW OF CL.>FI/PSI82C3/
CASE/PSI33A1/<SER. CL..IN/PSI82D2/<ROW OF CL.>ELSE/PSI82D3/<SER. CL.>FI

ENTRY PSI82D2

IF (.NOT.,SAMEYP(N(I),INTREAL,REF))
1 CALL ERROR(T10HINDEX OF A CASE STATEMENT IS NOT A NUMERICAL QU
  ANTITY.)
IF (.NOT.,(TYPE(I).EQ.9RNONSTORED)) GO TO 183
P=P+1
ITEM=CONVEPT(P,2R T)
CALL OUTDOA4 (Z, OPERAND(I), 1H+,ITEMP, 10H, ) 8980
CALL OUTDOV (P, REF, OUTCODE, Z, 1000-Z) 8990
I=I+1 9000
CODELOC(K)=7+1 9010
CALL OUTDOA4 (Z, ITEMP, 6H < 0, 0, 10H, ) 9020
CALL OUTDOA4 (Z, ITEMP, 6H > 0, 0, 10H, ) 9030
CALL OUTDOA4 (Z, ITEMP, 6H + 0, 0, 10H, ) 9040
CALL OUTDOA3 (Z, ITEMP, 10H+RJ(30+47), 10H, RJ, ) 9050

T(P)<0 LABEL ELSE::;
T(P)> NUMBER OF CLAUSES IN ROW1 LABEL ELSE::;
T(P)+ LABEL IF = T(P), (SEE PSI8203 FOR COMPLETION.) 9080
T(P)+RJ(30+47), RJ, 9090
CAUSES A JUMP TO A TABLE LEADING TO THE APPROPRIATE CLAUSE OF THE 9100
CASE STATEMENT, NOTE THAT STORAGE(BLOKNUM) IS USED TO RESET P TO THE 9110
SAME LEVEL FOR EACH CLAUSE.

THE SAVING OF CODELOC(K) IN WHAT FOLLOWS IS PERFORMED FOR COMPATIBILITY 9130
WITH PSI33A1 WHEN A<ROW OF CL.> APPEARS WITHIN A <COLLATERAL CLAUSE>.
9140

CALL BLOKEND(P, MODE)
K=K+1
CODELOC(K)=2
ENTRY PSI8203.
SEARCH HERE FOR THE MOST DEREFERENCED AND DEPROCEDURED (I.E., LOWEST) 9220
CLAUSE IN THE <ROW OF CLAUSE>, AND BRING THE REMAINING CLAUSES DOWN 9230
TO THAT LEVEL.

CALL BLOKEND(P, MODE)
CALL BLOKEND(P, MODE)
K=K+1
S=S-1
Z=Z+1
LINK1=MINIMUM(N(I))
RULES PSI183A AND PSI183B FOR <ROW OF CLAUSE> CREATE A LIST OF LABELS 9300
ON TOP OF THE OPERAND STACK, EACH ELT. OF THE LIST CONTAINS IN SEQUENCE 9310
(1) A POINTER TO THE NEXT ELT., 9320
(2) THE LABEL WHERE THE UNITARY CLAUSE BEGINS, 9330
(3) THE LABEL WHERE DEREFERENCING AND DEPROCEDURING 9340
OF THAT CLAUSE OCCURS, 9350
(4) THE LOCATION IN THE CODE OF (3) WHERE ESCAPE IS 9360
MADE TO THE CODE DIRECTLY FOLLOWING THE CASE 9370
STATEMENT, (SEE BALANCE SUBROUTINE.) 9380

ITEMP1=OPERAND(I)
ITEMP2=N(I)
CALL BALANCE (LINK1,ITEMP1,ITEMP2,S,P,Z,OUTCODE)
N(I)=LINK1

9440
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 OUTCD02 (Z,ITEM,10H1,      //
ITEMP=CODELOC(K)
K=K-1
OUTCODEITEMP1+2,3)=ITEMP
THE LOCATION OF THE CASE STATEMENT JUMP TABLE IS PUT INTO THE JUMP COND
ITEM02=OPERAND(I)
139 CALL OUTCD02 (Z,CONT(ITEMP2+2),10H*,       //
ITEMP2=LINKCONT(ITEMP2),1)
IF (ITEMP2,6T,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 OUTCD02 (Z,ITEM,10H1       //
OUTCODE(ITEMP1,3)=ITEMP
OUTCODE(ITEMP1,1,3)=CONVERT(DYTE( CONT(OPERAND(I)),10),2R )
OUTCODE(ITEMP1,1,5)=ITEMP
GO TO 184
ENTRY PSIA204
FURTHER BALANCING TAKES PLACE HERE BETWEEN THE <ROW OF CLAUSE> AND THE
<SERIAL CLAUSE>.
AFTER SAVING THE BLOCK VALUE, DO THE BLOCK END BOOKKEEPING.

CALL RLOKEND(P,MODE)
P=P+1
ITEMP4=CONVERT(P,2R T)
IF OPERAND(I),EQ,ITEMP4) GO TO 140
CALL OUTCD04 (Z,OPERAND(I),1H,ITEMP4,10H,       //
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 DEPRO
IF HIGHER, DEREFERENCE AND DEPROCEDURE THE <SER. CL.> THEN INSERT TH
FOR JUMPING OUT OF THE <ROW OF CL.> ROUTINES.

IF ().NOT.SAME(N(I-1),N(I),REF) GO TO 141
IF (REF,EQ,0) GO TO 142
GO TO 144
141 IF (SAME(N(I),N(I-1),REF) GO TO 145
CALL ERROR(1108) 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=S+1
ITEMP=CONVERT(S,2R L)
CALL OUTCD2(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 OUTCD2(Z,ITEMP,10H1) //
TYPE(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
COA%. NOTE THAT VALUE RETURN IS USED TO CARRY THE VALUES OF EACH CLAU

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

CALL OUTEVAL(P,REF,OUTCODE,Z,1000-Z)
CALL OUTCD2(Z,TYPE(I),10H1) //
N(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 OUTCD2(Z,ITEMP,10H1) //
CALL OUTEVAL(P,REF,OUTCODE,Z,1000-Z)
CALL LOOP82D(OPERAND(I-1),ITEMP,OUTCODE)
CALL OUTCD2(Z,ITEMP,10H1) //
GO TO 143

ENTRY PSI82C3
   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

54
TYPE(I)=6&STORED
$=S+1
ITEMP#CONVERT(S,2R L)
CALL OUTCOA2 (Z,ITEMP1,10H, //
ITEMP=CODELOC(K)
K=K+1
OUTCODE(ITEMP+2,3)=ITEMP
ITEMP3=OPERAND(I)
ITEMP=ITEMP3
146 CALL OUTCOA2 (Z,CONT (ITEMP3+2),10H, //
ITEMP3=LINK(CONT(ITEMP3),1)
IF (ITEMP3,GT,0) GO TO 146
S=S+1
ITEMP=CONVERT(S,2R L)
CALL OUTCOA2 (Z,ITEMP1,10H, //
OUTCODE(ITEMP,3)=ITEMP1
ITEMP=ITEMP+1
OUTCODE (ITEMP,3)=CONVERT(IYRE (CONT(OPERAND(I)),10),2R )
OUTCODE (ITEMP,5)=ITEMP1
P=P+1
ITEMP=CONVERT(P,2R T)
CALL OUTCOA3 (Z,SHD ,ITEMP,10H, //
S=S+1
ITEMP=CONVERT(S,2R L)
CALL OUTCOA2 (Z,ITEMP,10H, //
CALL LOOP&2D (OPERAND(I),ITEMP1,OUTCODE)
TYPE(I)=6&STORED
OPERAND(I)=ITEMP
RETURN

IF/PSI33A1/SEL/SERIAL CL./THEN/PSI82A2/SEL/SERIAL CL./FI/PSI82B3/
10990
11010
11020

ENTRY PSI82A2
2 CALL EXPOM(110HIF STATEMENT WITH NON-BOOLEAN TEST.
1 IF (.NOT.,SAMEQYI(N(I),900L,REF))
1040
1050
1060
1070
1080
1090
1100
1110
1120
1130
1140

CODELOC CARRIES LOCATION TO INSERT FIXUP LABEL HERE.

148 CALL BLOKEND(P,MODE)
11180
11190
11200
11210
11220
11230

RETURN
ENTRY PSIB2A3
IF (OPFRAND(I).EQ.ITEMP) GO TO 149
ITEMP=CONVERT(IBYTE(STORAGE(BOKNUM(I)),101+1,2R T)
CALL OUTCD2 (2,OPFRAND(I),1H+,ITEMP,10H, //)
TYPE(I)=6RSTORED
OPFRAND(I)=ITEMP
149 CALL OUTCD2 (?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 OUTCD2 (?ITEMP,10H+)
OUTCODE(ITEMP1,3)=ITEMP
GO TO 148
ENTRY PSIB2A4
CALL BLOKEND(P,MODE)
P=P+1
ITEMP=CONVERT(P,2R T)
IF (OPFRAND(I).EQ.ITEMP) GO TO 150
CALL OUTCD2 (2,OPFRAND(I),1H+,ITEMP,10H, //)
150 K=K-1
BALANCING OF CONSEQUENCE AND ALTERNATIVE
IF (.NOT.SAMEYP(N(I-1),N(I),REF)) GO TO 151
IF (REF,EQ.0) GO TO 152
GO TO 153
151 IF (SAMEYP(N(I),N(I-1),REF)) GO TO 155
CALL ERROR(110THE CONSEQUENCE AND ALTERNATIVE OF AN IF STATEM
ENT DO NOT HAVE THE SAME A POSTERIORI MODE.
)
NO BALANCING NEEDED.
152 S=S+1
ITEMP1=CONVERT(S,2R L)
CALL OUTCD2 (2,ITEMP1,10H+)
ITEMP2=CODELOC(K)
OUTCODE(ITEMP2,1)=ITEMP1
K=K-1
GO TO 154
LOWER THE CONSEQUENCE TO THE LEVEL OF THE ALTERNATIVE, AND CAUSE THE A
TIVE TO SKIP OVER THE LOWERING CODE.
153 N(I-1)=N(I)
CALL OUTCD2 (?0,10H+, //)
ITEMP=7
S=S+1
ITEM1=CONVERT(S,2R L)
ITEM2=CODELOC(K)
K=K+1
CALL OUTCDA2 (Z,ITEM1,10H) //
OUTCODE(ITEM2,1)=ITEM1
CALL OUTEVAL (P,REF,OUTCODE,Z,1000-Z)
S=S+1
ITEM1=CONVERT(S,2R L)
CALL OUTCDA2 (Z,ITEM1,10H) //
ITEM2=CODELOC(K)
K=K+1
OUTCODE(ITEM2,1)=ITEM1
GO TO 154

I=I-1

RETURN

; LOWER THE ALTERNATIVE, THEN INSERT JUMP CODE FOR THE CONSEQUENCE.
; 155 CALL OUTEVAL (P,REF,OUTCODE,Z,1000-Z)
S=S+1
ITEM1=CONVERT(S,2R L)
CALL OUTCDA2 (Z,ITEM1,10H) //
ITEM2=CODELOC(K)
K=K+1
OUTCODE(ITEM2,1)=ITEM1
GO TO 154

; ENTRY PSI8283
; INSERT ALTERNATIVE CODE AND PROVIDE A JUMP TO AVOID THIS CODE.
ITEM=STORAGE(BLOKNUM)
P=BYTE(ITEM,1D)+1
ITEM1=CONVERT(P,2R T)
CALL BLOKEND (0,0)
CALL BLOKSN (0,0)
IF (OPERAND(I),EQ,ITEM1) GO TO 156
CALL OUTCDA4 (Z,OPERAND(I),1H*,ITEM1,10H, //
YPEP(I)=6RSTORED
OPERAND(I)=ITEM1
156 CALL OUTCDA2 (Z,0,10H*, //
S=S+1
ITEM2=CONVERT(S,2R L)
K=K+1
ITEM2=CODELOC(K)
K=K+1
CALL OUTCDA2 (Z,ITEM2,10H) //
OUTCODE(ITEM1,3)=ITEM2
CALL OUTCDA3 (7,5H0*,ITEM1,10H, //
S=S+1
ITEM2=CONVERT(S,2R L)
OUTCODE(7-2,1)=ITEM2
CALL OUTCDA2 (7,ITEM2,10H) //
RETURN
ENTRY PSI84A

C IFST TO SFE 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
C

11770
11780
11790
11800
11810
11820
11830
11840
11850
11860
11870
11880
12000
12010
12020
12030
12040
12040
12050
12060
12070
12080
12090
12100
12110
12120
12130
12140
12150
12160
12170
12180
12190
12200
12210
12220
12230
12240
12250
12260
12270
12280
12290
12300
12310
12320
JUMP TABLE BEGINS.

P=IYFTE(STORAGE(BLCKNUM),10)
ITEM=CONVET(I+P,2R T)
IF (OPERAND(I).EQ.ITEM) GO TO 157
CALL OUTCODA (2,OPERAND(I),ITEM,0H,10H, /*)
157 CALL OUTCODA3 (2,10HRETURN*,RJ,8H(30*47),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 OUTCOME CARDS PUT IN BY PSI33A1 DURING PSI5A
ITEM2=CODELOC(K-2)
CALL OUTCODA2 (ITEM2,ITEM3,0H1,10H1 /*)
CODELOC(K-1)=ITEM3
CODELOC(K-2)=ITEM2-1
158 CALL STPIND (CODELOC(K-1),ITEMP+1)
OPERAND(I)=ITEMP1

STORED ON OPERAND(I) IN A LIST CELL.
S=S+1
ITEM3=CONVERT(S,2R L)
CODELOC(K-1)=ITEM3
CALL OUTCODA2 (2,ITEM3,0H1,10H1 /*)
RETURN

THE LABEL AT WHICH CODE BEGINS FOR EACH ROW IN A <ROW OF CLAUSE> IS
ENTRY PSI5A3A

DO PSI5A4A WORK FIRST, THEN PIECE TOGETHER THE COMPILE-TIME REPRESENTATION OF <ROW OF CLAUSE>

P=IYFTE(STORAGE(BLCKNUM),10)
ITEM=CONVET(P+1,2R T)
IF (OPERAND(I).EQ.ITEM) GO TO 159
CALL OUTCODA (2,OPERAND(I),1H*,ITEM,10H, /*)
159 CALL OUTCODA3 (2,10HRETURN*,RJ,8H(30*47),10HRJ, /*)
ITEM=IFETCH(4)
CALL STRIND (CODELOC(K-1),ITEMP+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
ITEM=IFETCH(1)
CALL SETINS (N(I)),ITEMP,2)
I=I-1
ITEMP=IFETCH(1)
CALL SETIND (N(I),ITEMP,2)
CALL SETINE (ITEMP,ITEMP,1)
CALL PUTEIN (2,ITEMP,10)
M(I)=ITEMP
RETURN

ENTRY PSI83D
ITEM=IFETCH(1)
CALL SETINS (N(I),ITEMP,1)
I=I-1
CALL SETIND (I(I),ITEMP,2)
M(I)=ITEMP
CALL PUTEIN (1+BYTE(CONT(N(I+1)),10),ITEMP,10)
CALL STIND (operand(I+1),operand(I))
RETURN
ENTRY PSI85A2

INSERT JUMP AROUND <POW OF CLAUSE> CODE, THEN GO BLOCK END ROUTINE.

K=K-2
ITEMP=CODELOC(K)-2
ITEMP=ITEMP+1
DO 130 I=ITEMP,ITEMP1
160 OUTCODE(I1,I2)=2007/
CALL OUTCODE2 (CODELOC(K),CODELOC(K+1),10H,, )
GO TO 16A
ENTRY PSI14882

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.

ITEMP=CONT(N(I))
ITEMP2=CONT(N(I-1))
P=P+1
ITEMP3=BYTE(ITEMP,10)
IF (ITEMP3,EO,1BYTE(ITEMP2,12)) GO TO 161
CALL ERROR (110,THE <COLLATERAL CLAUSE> AND ITS A POSTERIORI MODE
IS NOT THE SAME LENGTH.)
RETURN
161 ITEMP=LINK(ITEMP,1,3)
IF (ITEMP4,EO,ARRAY) GO TO 162
IF (ITEMP4,EO,ARR) GO TO 164
CALL ERROR (110,THE A POSTERIORI MODE OF A <ABOUT CAST> WITH A <CO
LATERAL 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.

152 CALL OUTCDA4 (7, 7HSAVEU, LMCURR, 10H+1#U, SAVEU, 10H, //)

CREATE A RUN-TIME ARRAY: STEP 11 GENERATE THE DESCRIPTOR.

ITEMP4=CONVERT (ITEMP3, 2R)
CALL OUTCDA6 (7, LMCURR, 1H+, ITEMP4, 1H+, LMCURR, 10H, //)
CALL OUTCDA3 (7, 10H0, 10H0000000000, 10H0000 + U1, 10H, //)
CALL OUTCDA3 (7, 10H0+2, + (U1, 10H0+17#1), U1, 10H#U1, //)
CALL OUTCDA3 (7, 10H030000000100, 10H00000000001, 10H#U1, //)
CALL OUTCDA (7, CONVERT (ITEMP3, 2R), 10H#(UI18=35, 8H), //)

STEP 21 CALL LOWER, AFTER GENERATING CIRCULAR LIST FOR LOWEST MODE.

ITEMP4=MINIMUM (N(I))
ITEMPS=IFETCH(1)
CALL SETIND (ITEMPS, ITEMP5, 1)
CALL SETIND (ITEMPS, ITEMP5, 2)
CALL LOWER (ITEMP3, ITEMP5, N(I), OPERAND(I), Z, P)

STEP 31 COMPLETE WORK ON THE RUN-TIME DATA ELEMENT.

153 ITEMP4=CONVERT (P, 2R T)
CALL OUTCDA3 (7, 10HP0PU, U + ITEMP4, 10H, P0PU, //)
I=I-1
OPAND(I)=ITEMP3
RETURN

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

164 CALL OUTCDA4 (7, 7HSAVEU, LMCURR, 10H+1#U, SAVEU, 10H, //)
CALL OUTCDA6 (7, LMCURR, 1H+, CONVERT (ITEMP3, 2R ), 1H+, LMCURR, 10H, //)
CALL LOWER (ITEMP3, LINK (ITEMP2, 2), OPERAND(I), Z, P)
GO TO 163

ENTRY PSI49A1

THE MORE 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 CONVENTION OF LEAVING THE DESTINATION OF AN ASSIGNMENT AS ITS VALUE AND BECAUSE OUR PSI33A2 RESETS P.

M(I)=0
OPAND(I)=0
RETURN

ENTRY PSI48A2

P = P + 1
ITEMP=CONVERT (P, 2R T)
IF (.NOT. (ITEMP.EQ.OPERAND(I))) GO TO 165
   CALL OUTCD4r (Z,OPERAND(I),1H,ITEMP,10H, //)
165 IF (SAMETFY(N(I)-1),N(I),REF)) GO TO 166
   CALL ERROR (110H THE A PRIORI MODE OF THIS <MODE CAST> DOES NOT MA
170 ITS A POSTERIORI MODE.)
RETURN
166 CALL OTEVAL (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 PSI46A2

DEREFERENCING AND DEPROCEDUREING MAY TAKE PLACE HERE.

P=P+1
ITEMP=CONVERT(P,2R T)
IF (ITEMP.EQ.OPERAND(I)) GO TO 167
   CALL OUTCD4r (Z,OPERAND(I),1H,ITEMP,10H, //)
167 IF (SAMETFY(N(I),N(I)-1),REF)) GO TO 168
   CALL ERROR (110HA PRIORI MODE OF THE <MODE CAST> DOES NOT MATCH ITS
1 A POSTERIORI MODE.)
RETURN
168 CALL OTEVAL (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 (1+IBYTE(CONT(N(I))),10,N(I),10)
ITEMP1=1
   GO TO 171
ENTRY PSI16B

ITEMP=ITEMP1
171 ITEMPE=IFETCH(1)
   CALL SETIND (N(I),ITEMP,ITEMP1)
   CALL SETIND (3PRC,ITEMP,3)
   N(I)=ITEMP

ENTRY PSI16B
RETURN
ENTRY PSI15B1
K=K+1
CODELOC(K)=Z
RETURN
ENTRY PSI16C
ITEMP=IFETCH(1)
CALL SETING(N(I),ITEMP,2)
CALL SETIND(N(I-1),ITEMP,1)
CALL PUTIND(I,BYTE(CONT(N(I-1)),10),N(I),10)
I=I-1
N(I)=ITEMP
RETURN
ENTRY PSI77A1
IN THE PROCEDURE CALL, THE <PRIMARIES> MUST BE LOWERED TO ITS FIRST MOD
A PROCEDURE WITH PARAMETERS.

NI=N(I)
ITEMP=LOCF(ITEMP4)
IF (NI,GT,0) GO TO 173
172 CALL ERROR(111) PROCEDURE CALL ATTEMPTED ON A NON-PROCEDURE <PRIMARIES>.
RETURN
173 ITEMPS=CONT(N(I))
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 ITEMPS=IFETCH(1)
CALL SETING(ITEMP2,ITEMP5,3)
CALL SETIND(ITEMP5,ITEMP7,2)
ITEMP3=ITEMPS
NI=LINK(ITEMP2)
GO TO 173
175 IF (ITEMP1,NE,3RPAC) GO TO 172
IF (LINK(ITEMP2,GT,0) GO TO 176
ITEMP2=3RPAC
GO TO 174
176 REF=ITEMPS
P=P+1
ITEMPS=CONVERT(P,2R T)
IF (OPERAND(I),EQ,ITEMPS) GO TO 177
CALL OUTCDA4(2,OPERAND(I),1H,ITEMP4,16H, //)
OPERAND(I)=ITEMPS
TYPE(I)=64RSTORE
177 N(I)=NI
CALL OUTFVAL (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

I=I+1
S=S+1
ITEMP4=CONVERT(S,2R L)
N(I)=ITEMP4
CALL OUTCODA4 (Z,0,0,ITEMP4,10H, //)
 Tipo(I)=Z
CALL SECTOR (P+1,TYPE(I),2)
RETURN

ENTRY PSI78B1

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

178 ITEMP3=LINK1 TYPE(I-1),2)
ITEMP3=50
ITEMP1=CONVERT4ITEMP3,2R T)
IF (ITEMP3.EQ.OPERAND(I)) GO TO 179
CALL OUTCODA4 (Z,OPERAND(I),1H,ITEMP1,10H, //)
179 P=ITEMP1
CALL OUTCODA3 (7,10H RETURN+ R1,10H(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=CONVERT4S,2R L)
CALL OUTCODA2 (Z,ITEMP2,10H, //)
N(I)=ITEMP2
CALL SECTOR (ITEMP1,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), I)
I=I-1
RETURN

ENTRY PSI77A2

OPERAND(I) CONTAINS THE LIST OF ENTRY LABELS AND DATA TYPES FOR EACH
ACTUAL PARAMETERS. 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 PROCEDURE, AT END OF
R 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
ITEM1=CONVERT(S,2R L)
OUTCODE(ITEM1)=ITEM1
OUTCODE(ITEM1)=10H+
CALL OUTCDA2 (Z,ITEM1,10H1 //)
P=P+1
N=CONVERT(P,2R I)
CALL OUTCDA4 (Z,7HSAVE U,,LHCU,5H+1#U,,10HSAVE U, //)

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

ITEM2=CONT(ITEM1)
ITEM2=ITEM2+ITEM5
IF (SAMEYPE(LINK(ITEM2,2), LINK(ITEM4,2), REF)) GO TO 181
CALL ERROR(110 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 OUTCDA9 (Z,10HSAVE RETUR,10H, J+2#RETUR,3HRN,,CONT(ITEM1+1), 1
10H+, POP RETU,10H, //)
CALL OUTCDA1 (P,REF,OUTCODE,Z,1000-2)
CALL OUTCDA4 (Z,LHCURR,3H+1#LHCURR,10H, //)
CALL OUTCDA4 (Z,N1,1H#,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 (110H L LEGAL PROCEDURE CALL--THE NUMBER OF ACTUAL AND F
10RMAL 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, 10HENVIROMEN, 10HT SAVE, //)
CALL OUTCDA6 (Z, 10HSAVE RETUR, 2HN, OPERAND(I), 10H(0=17) A JUM, 10HP, J
12=RETU, 10HRRN, //)
CALL OUTCDA5 (Z, 8HJUMP),, 10HP0P RETURN, 10HRESTORE EN, 10HENVIRONMEN
1T, 10H, //)
CALL OUTCDA3 (Z, 8HVVALUE A, OPERAND(I), 10H, //)
TYPE(I)=6RSTORED 
P=P+1 
RETURN

ENTRY PSIT3A1

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

P=P+1 
ITEMP=CONVERT(P, 2R T)
S=S+1
ITEMP=CONVERT(S, 2R L)
CALL OUTCDA5 (Z, 1H*, ITEMP, 1H=, ITEMP, 10H, //)
CALL OUTCDA6 (Z, 10HPROC LEVEL, 1H=, ITEMP, 10H(18-35), //)

THE MASK ENVIRONMENT ROUTINE TESTS DELEVEL. IF DELEVEL = 0, THEN NO
RESTORATION. OTHERWISE THE ENVIRONMENT SAVE TABLE IS DUMPED ONTO THE
NAME TABLE AT THE LEVEL GIVEN BY DELEVEL.

I=I+1
OPERAND(I)=ITEMP 
TYPE(I)=6RSTORED 
N(I)=7

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

CALL PROCGRN(P, 0)

; ENTERS THE HASH TABLE AND DOES BLOCK BEGIN BOOKKEEPING

RETURN

16540
16550
16560
16570
16580
16590
16600
16610
16620
16630
16640
16650
16660
16670
16680
16690
16700
16710
16720
16730
16740
16750
16760
16770
16780
16790
16800
16810
16820
16830
16840
16850
16860
16870
16880
16890
16900
16910
16920
16930
16940
16950
16960
16970
16980
16990
17000
17010
17020
17030
17040
17050
17060
17070
17080
17090
17100
17110
17120
ENTRY PSI73A2

ITEM=IFETCH(1)
CALL SETIND(3,RG,ITEM,3)
CALL SETIND(N(I),ITEM,2)
CALL OUTODA4(Z,OPERAND(I),10H -> VALUE,13H(RETURN),10H
17130
17140
17150
17160
17170
17180
17190
17200
17210
17220
17230
S=S+1
ITEM1=CONVERT(S,2R L)
I=1
ITEM2=N(I)
OUTCODE(IITEM2,1)=ITEM1
CALL OUTODA2(Z,ITEM1,10H)
N(I)=ITEM
CALL PRCEND(P,0)
RETURN
17240
17250
17260
17270
17300
17310
17320
17330
17340
17350
17360
17370
17380
17390
17400
17400

END
SUBROUTINE OUTEVAL (P,REF,OUTCODE,Z,MAX)
 INTEGER P,REF,OUTCODE,Z,CONT,CONVERT
 DIMENSION OUTCODE(MAX,5)
 ITEM=CONVERT(P,2R T)
 101 ITEM3=LINK(CONT(REF),2)
 102
 C REF POINTS TO THE LIST CREATED BY SAMETYP CONTAINING INFORMATION
 C ABOUT THE CORRECT SEQUENCE FOR DEPROCEDUREING AND DEREFERENCEING
 C THE EXPRESSION CURRENTLY BEING EVALUATED.
 C IF (ITEM3.EQ.0) RETURN
 C 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 )
 180 CALL OUTODA4(Z,ITEM4,10H=NUMBER OF,10HPARAMETERS,8H, //
 190 P=P-LINK(CONT(REF),3)
 200 ITEM=CONVERT(P,2R T)
 210 CALL OUTODA4(Z,10HSAVE RETUR,3HN,,10HITEM4P= register,10H30047), //
 220 230
 C CONTROL IS PASSED TO THE PROCEDURE9
 C CALL OUTODA2(Z,10HJ+2RETURN,10H, RJ, //
 240 CALL OUTODA5(Z,10HPOPPAR RETURN,10HVALUE RETU,3HRN*,ITEM,10H, 250 1 //
 260 CALL OUTODA2(Z,10HPOPOP VALUE,10HRETUR, //
 270 280 290 300 310 320 330 340
 C THE NUMBER OF REFERENCE LEVELS IS FOUND IN LINK 3 OF THE WORD POINTED
 C TO BY REF.
C 102 ITEM4=LINK(CONT(REF),3)
GO TO 104
103 ITEM4=ITEM4-1
104 CALL OUTCD45 (Z1H,ITEMP2,TEMP1,10H, //)
IF (ITEMP4.GT.0) GO TO 107
REF=LINK(CONT(REF),1)
GO TO 101
C END
INTEGER FUNCTION MINIMUM(NI)
LOGICAL SAMEYPE
INTEGER REF,CONT
ITEMP1=CONT(NI)
LINK1=LINK(ITEMP1,2)
ITEMP=LINK(ITEMP1,1)
101 ITEMP1=CONT(ITEMP)
LINK2=LINK(ITEMP2,2)
IF (SAMEYPE(LINK1,LINK2,REF)) GO TO 102
C AT 5, LINK2 IS THE LOWER DATA TYPE.
C IF (SAMEYPE(LINK2,LINK1,REF)) GO TO 103
CALL ERROR (52)
RETURN
102 LINK1=LINK2
103 ITEMP=LINK(ITEMP1,1)
IF (ITEMP.GT.0) GO TO 101
MINIMUM=LINK1
RETURN
C END
SUBROUTINE BALANCE (LINK1,ITEMP1,ITEMP2,S,P0,Z,OUTCODE)
LOGICAL SAMEYPE
INTEGER S,P,Z,OUTCODE,CONT,P0
ITEMP=LINK(CONT(ITEMP2),2)
11 T S=S+1
ITEMP=CONVERT(S,2R L)
P=P0+1
CALL STRIND (ITEMP,ITEMP1+2)
CALL OUTCD2 (Z,ITEMP,10H, //)
CALL OUTCD3 (Z,10HSAVE RETUR,10HN,JZ=RETU,10HN, //)
CALL OUTCD2 (Z,CONT(ITEMP1+1),10H, //)
CALL OUTCD3 (P,REF,OUTCODE,Z,10HH-Z)
CALL STRIND (Z+1,ITEMP1+3)
C HERE, THE ESCAPE ADDRESS IS PUT INTO THE THIRD LIST ELT. DATUM.
C SEE PSIB2D3 FOR DETAILS. NOTE THAT THE ROWS OF CLAUSE USE TICODELOCKK
C AS THE VALUE RETURN ABOVE.
C CALL OUTCD2 (Z,0,10H,, //)
ITEMP1=CONT(ITEMP1)
ITEMP2=LINK(CONT(ITEMP2),1)
IF (ITEMP1.GT.0) GO TO 101
IF (ITEMP2.EQ.0) RETURN
    CALL ERROR(110ERROR IN COMPILER'S BALANCE SUBROUTINE.
    STOP
RETURN
END
SUBROUTINE LOOP82D (LINK, LABEL, OUTCODE)
DIMENSION OUTCODE(5000, 6)
INTEGER OUTCODE
    ITEMP=INHALT(LINK+3)
    OUTCODE(ITEMP, 1)=LABEL
    ITEMP=LINK(CONT(ITEMP), 1)
IF (ITEMP .EQ. 0) RETURN
GO TO 101
END

SUBROUTINE LOWER(LENGTH, STANDARD, ROW, OPRANDI, Z, P0)
INTEGER ROW, STANDARD, OPRANDI, Z, P0, P, CONT, REF, CONVERT

LOGICAL SAFETY
LENGTH GIVES THE NUMBER OF FIELDS TO LOWER, STANDARD IS THE A POSTERIORI
STRUCTURE, ROW IS THE A PRIORI MODE STRUCTURE, OPRANDI IS THE LIST OF
CLAUSE ENTRY LABELS.

ITEMP3=CONVERT(P, 2R  T)
    IG=OPRANDI
    I1=STANDARD
    I2=ROW
DO I2=1, LENGTH
    ITEMP1=CONT(I1)
    ITEMP2=CCONT(I2)
    LINK1=LINK(ITEMP1, 2)
    LINK2=LINK(ITEMP2, 2)
    IF (SAMEYPEP(LINK2, LINK1, REF)) GO TO 101
    CALL ERROR1(110THE FOLLOWING <ROW OF CLAUSE> FIELD DOES NOT
    MATCH ITS A POSTERIORI MODE.
        CALL ERROR1(110, CONVFR, 2R )
    END
    FIND LABEL OF NEXT CLAUSE IN THE ROW, GENERATE A RETURN JUMP TO IT, T
    CALL OUTEVAL TO LOWER IT.
        CALL OUTCDA3 (Z, 10HSAVE RETUR, 10HNJ+2, RETU, 10HRN,  //)
        CALL OUTCDA2 (Z, CONT(I1)+1, 10H, ), //)
        CALL OUTEVAL (P, REF, OUTCODE, Z, 1000-Z)
        CALL OUTCDA3 (Z, 6H+1+U, , ITEMP3, 10H-U,  //)
101 I0=CONT(I0)
    I1=LINK(ITEMP, 1)
    I2=LINK(ITEMP, 2)
    CALL ERROR1(110ERROR, CONVFR, 2R )
    RETURN
END

INTEGER FUNCTION IHAMNNAME, MODE)

DIMENSION NAMES1(197, 2), NAMES2(197, 2), HASH1(197, 2), HASH2(197, 2)
1, STORAGE(50), PSTORE(50)
INTEGER HASH1, HASH2, STORAGE, BLOKNUM, PLEVEL, PRCOUNT, BLCOUNT, PSTORE
10 CONVERT
LOGICAL SAME_TYPE
20 ENTRY INITIAL
30 CALLED ON ENTRY TO THE COMPILATION OF A NEW PROGRAM.
40 PLEVEL=0
50 BLOKNUM=0
60 PRCOUNT=0
70 MBCOUNT=0
80 HPBCOUNT=0
90 L=IFETCH1(0)
100 INITIALIZES ALL COMPILE TIME LISTS
110 RETURN
120 ENTRY FINAL
130 CALLED WHEN COMPILER REQUESTS TALLY OF NAMES IN NAMES AND PROCS TABLE
140 NAME=MBCOUNT
150 MAXIMUM NAMES IN STATIC ENVIRONMENT
160 MODE=HPBCOUNT
170 MAXIMUM NO. OF NAMES IN PROCEDURE ENVIRONMENT.
180 RETURN
190 ENTRY BLOKGEN
200 BLOKNUM=BLOKNUM+1
210 STORAGE(BLOKNUM)=BCOUNT
220 CALL SETDIR(BCOUNT, STORAGE(BLOKNUM), 2)
230 CALL SETDIR(NAME, STORAGE(BLOKNUM), 3)
240 L=IFETCH2(0)
250 COMPILE-TIME LISTS BOOKKEEPING
260 RETURN
270 ENTRY BLOKEND
280 L=BCOUNT-STOREAGE(BLOKNUM)-1
290 IF (L.LT.0) GO TO 103
300 DO 102 I=0, L
310 J=NAME1(BLCOUNT-I, 1)
320 IF (J.EQ.0) GO TO 102
330 102 BLCOUNT=STORAGE(BLOKNUM), AND, 7777778
PROC.LINK(STORAGE(BLOKNUM),2)
NAME=LINK(STORAGE(BLOKNUM),3)
550
BLOKNUM=BLOKNUM-1
560
104 L=IPATCHI(6)
570
CALLS THE LIST GARBAGE COLLECTOR.
580
RETURN
ENTRY PROC&GN
590
PRLEVEL=PRLEVEL+1
600
PSTORE(PRLEVEL)=PROCOUNT
610
call SETDIR(NAME,PSTORE(PRLEVEL),2)
620
GO TO 101
630
ENTRY PROCEND
640
DELETES COMPILE TIME NAMES IN CURRENT PROCEDURE BLOCK, BUT LEAVES THE
650
TIME LOCATIONS INTACT.
660
L=PROCOUNT-1-LINK(PSTORE(PRLEVEL),1)
700
NAME=LINK(PSTORE(PRLEVEL),2)
710
IF (L.LT.0) GO TO 106
720
DO 105 I=0,L
730
J=NAMES2(PROCUE,1,1)
740
105 HASH2(J,1)=0
750
106 PRLEVEL=PRLEVEL-1
760
GO TO 104
770
ENTRY IHASHA
780
LOOK UP NAME AND MODE, OR LABEL AND LIST OF LOCATIONS TO BE FILLED, AND
790
RUN-TIME ALIAS PRECEDED BY ".
800
LASTUSE=0
810
I=NAME
820
IF (PRLEVEL.EQ.0) GO TO 111
830
DO 110 L=0,196,1
840
IA=MOD(I+47*L,197)+1
850
IF (IA.NE.0) GO TO 108
860
IF (LASTUSE.GT.0) GO TO 107
870
GO TO 111
880
107 IHASHA=CONVERT(LASTUSE,2*P)
890
MODE=LOCF(NAMES2(LASTUSE,2))
900
RETURN
910
108 IF (IA.NE.NAME) GO TO 109
920
LASTUSE=HASH2(IA,2)
930
109 CONTINUE
940
110 CONTINUE IF(LASTUSE.GT.0) GO TO 107
950
NAME IS NOT WITHIN PROCEDURE ENVIRONMENT.
960
111 00 115 L=0,196,1
970
112
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  IHASHA=CONVERT(LASTUSE,2R^V)
MODE=LOCF(1AMES1(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 IHASHB
; ENTER NAME AND MODE ON NAMES TABLE, AND RETURN RUN-TIME ALIAS PREFIXED
; I=NAME
IF (APPLEVEL.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
; ENTER THE NAME.
116  CONTINUE
CALL ERROR(110,COMPILE-TIME NAME TABLE OVERFLOW DURING A PROCEDURE DECLARATION.)
RETURN
117  PRCOUNT=PRCOUNT+1
IF (PRCOUT,GT.,PRCOUNT) PRCOUNT=PRCOUNT
HASH2(I,1)=NAME
HASH2(I,2)=PRCOUT
NAMES2(1AMES1,1)=I
NAMES2(1AMES1,2)=MODE
MODE=LOCF(1AMES1(1AMES1,2))
FOR USE BY LABELS.
IHASHB=CONVERT(PRCOUNT,2R^P)
RETURN
118  DO 119 L=0,196,1
I=MOD(I+47*L,197)+1
IA=HASH1(I,1)
IF (IA,NE.,0) GO TO 120
119  CONTINUE
CALL ERROR(110,COMPILE-TIME NAME TABLE OVERFLOW OUTSIDE OF A PROCEDURE DECLARATION.)
RETURN
120  BLCOUNT=BLCOUNT+1
IF (BLCOUNT,GT.,MBCOUNT) MBCOUNT=BLCOUNT
THIS FUNCTION LOOKS UP OPERATORS IN THE NAME TABLE. WE ASSUME THAT AN OPERATOR NOT PRESENT ON THE TABLE IS EITHER INVALID OR WIRE IN TO THE RUN-TIME PILOT SYSTEM.

LASTUSE = 0
I = NAME
IF (LEVEL < 0) GO TO 124
127    L = 0, 196, 1
       IA = MOD((I+47*L, 197) + 1
       IF (IA .LE. 0) GO TO 121
       IF (LASTUSE.GT.0) GO TO 107
       GO TO 124
121    IA = HASH2(I, 2)

COMPARE OPERAND MOTES.

IF (LINK(CONT.NAME2(IB, 2)), I)
       IF (IA .NE. NAME) AND (.NOT.SAMEYPE(MODE, IB, REF)) GO TO 122
       LASTUSE = HASH2(I, 2)
       CONTINUE
122
123 CONTINUE

IF (LASTUSE.GT.0) GO TO 107
124 70 126 L = 0, 196, 1
       IA = MOD((I+47*L, 197) + 1
       IF (IA .LE. 0) GO TO 125
       IF (LASTUSE.GT.0) GO TO 112
       IA = HASH2(I, 2)
       RETURN
125    IA = HASH2(I, 2)
       IF (LINK(CONT.NAME2(IB, 2)), I)
       IF (IA .NE. NAME) AND (.NOT.SAMEYPE(MODE, IB, REF)) GO TO 123
       LASTUSE = HASH2(I, 2)
       CONTINUE
123 CONTINUE

IF (LASTUSE.GT.0) GO TO 112
RETURN
END