1970

A One Pass Algorithm for Compiling ALGOL 68 Declarations

Victor B. Schneider

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

Victor B. Schneider

CSD TR 53
October 1970
(Revised October, 1971)

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

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

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>List Processing Primitives Used by the Compiler</td>
</tr>
</tbody>
</table>
# FIGURES

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

1. Introduction

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

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

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

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

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, \( \psi_{25a1} \), \( \psi_{25a2} \), 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

"MODE <mode indicant> ="

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

\[ P_{25a} \] causes subroutine \( \psi_{25a3} \) to be called.

Thus, the ensemble of rules in a translation grammar may be thought of as an abstract representation of the program input to a compiler-compiler system.

In fact, our translation grammar is intended to be transferred to cards and read in to a compiler-compiler such as the one described in (6).

As can be seen in Appendix 2, the subroutines for the data declaration section of our translation grammar have been programmed in FORTRAN V (no less), and it is the operation and interaction of these compiler subroutines that is the main burden of this report.
II. Techniques Used in Translation

Representation of ALGOL 68 Structures and Data Types in the Compiler

ALGOL 68 structures and invented data types are represented as linked lists in the computer memory used by the compiler. Such a representation is, naturally, machine dependent, so we propose here to give examples of how these lists are represented as storage structures on the Purdue 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:

```plaintext
MODE $\text{FF}$ = PROC(INT,$\text{SS}$)REAL;
MODE $\text{SS}$ = STRUCT(REF $\text{SS}$ POINTER, $\text{QQ}$ VALUE);
MODE $\text{QQ}$ = UNION($\text{QQ}$, $\text{SS}$);
MODE $\text{RS}$ = REF $\text{SS}$;
MODE $\text{T}$ = [1:5, 1:6]REF $\text{SS}$;
```

Routines Used in Manipulating Compiler List Structures

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

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

In addition to the functions in Table 1, there is an integer function called IFETCH(K) and a logical function called SAMEI(K,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 SAMEI function is, as the name suggests, a function for comparing two compiler data structures to discover if they are of the same "type". Because of the design of ALGOL 68, two declarations may be of the same
"type" if one has any number of intermixed procedure and reference prefixes followed by a copy of the second declaration. Thus, for example, if parameter I pointed to a declaration, such as "PRSC REF REF PRSC REAL", and J pointed to "PRSC REAL", the 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 "PRSC REF REF". Thus, the list returned to K indicates how many levels of dereferencing and deproceduring must be applied to a datum of type I in order to yield a datum of type J.

The SAMETYP function also incorporates algorithms for looking up the lists assigned to mode indicators appearing in I or J and for replacing these mode indicators in either structure by pointers to these lists. No attempt is made to find reduced versions of structure declarations. However, when two structures are compared that are structurally equivalent as defined in the ALGOL 68 report, a link to the smaller of the two structures replaces the larger structure in the compiler mode table. This structure comparison and replacement algorithm is essentially the same as the one given by Koster (11) and discussed in Goos (9).

Intermediate Language Generated by the Compiler

Since this report is the beginning of an attempt at producing a formal specification of ALGOL 68 that is an alternative to the Van Wijngaarden and the Vienna notations (12), it was decided that the code produced by our compiler subroutines would be in a systems language rather than in the C.D.C. COMPASS assembly language. The systems language chosen is the PILOT language of W.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 mixing of PILOT and machine code where desirable in a program, the use of machine addresses for indirect referencing of variable names, and the addition of partial-word masking operators. In PILOT, commas are used to separate statements, expressions can consist of at most two operands separated by an operator, and assignment of value is to the right, rather than to the left, as in ALGOL 60 or FORTRAN. A typical sequence of PILOT statements might be the following:

\[
\begin{align*}
0 & + \text{INDEX REGISTER 1}, \\
U - 1 & + U,
\end{align*}
\]

\[
\text{LABEL 1: INDEX REGISTER 1} + 1 \rightarrow \text{INDEX REGISTER 1},
\]

\[
\text{INDEX REGISTER 1} > \text{SIZE} : \text{LABEL 2};
\]

\[
\text{LHCURR} + 1 \rightarrow [U + \text{INDEX REGISTER 1}],
\]

\[
\text{LABEL 2}:
\]

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 "LHCURR + 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.

"CODELAC" is the system name for the second stack, a one-dimensional array with index K. As its name implies, the CODELAC 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 CODELAC which enables us to produce reasonably efficient and non-redundant code in a single-pass compilation process. Of course, CODELAC 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 CODELAC 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 readily single-pass version of what is usually considered a multi-pass compiler system.
Criteria for Selecting Rule Forms in a Translation Grammar

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

<mode indicant> + $<name>$
<operator> + + | - | ... | + <name> +
<call> + <primary> (<actual parameters>)
<slice> + <primary> [<indexers>]
<base> + 0# TO <label>

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

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

does not mean the same thing as

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

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 prelude terminates at the first labeled statement or the first statement followed by a jump. To translate such a declaration prelude properly, the compiler needs to know at the beginning of each statement in a declaration prelude what
comes after that statement. Such knowledge could be gained by a separate pass or by requiring forward scans before each statement, but this extra work consumes compiler time that could be better spent doing translation. Hence, our version of the ALGOL 68 block (rules P_34 and P_35 in Appendix I) 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 I. In this case the syntax for <stowed declarer> initially took the form:

\[
<\text{stowed declarer}> \rightarrow \text{STRUCT} \langle \text{fields} \rangle \\
\mid [\langle \text{rows} \rangle \text{STRUCT} \langle \text{fields} \rangle ] \\
\mid [\langle \text{rows} \rangle \text{<nonstowed declarer>}] \\
\mid [\langle \text{rows} \rangle \text{<mode indicant>}]$
\]

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 I:
<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:

F_{29a}: <reference to actual declarer> + <actual mode
declarer> /\theta_{29a}/

F_{2a}: <nonstowed declarer> + REF/\psi_{2a}/ <mode declarer>/\theta_{2a}/

F_{30a}: <reference to mode global generator> + HEAP/\psi_{30a}/

    <actual mode declarer>/\theta_{30a}/

In the second example, the rules

F_{21a}: <virtual mode declarer> + <mode declarer>/\theta_{21a}/

F_{22a}: <actual mode declarer> + <mode declarer>/\theta_{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 \psi_{21a} and \psi_{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:

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

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

On the other hand, the rules

\[
P_{7a}: \text{<rowers> } + \text{<rower>}/\psi_{7a}/
\]
\[
P_{7b}: \text{<rowers> } + \text{<rowers>, <rower>}/\psi_{7b}/
\]
do nothing more than construct a run-time array descriptor in the order of appearance of the program.
III. Translation of Array and Structure Declarations

Preliminaries

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

During compilation, the compiler subroutines keep track of whether
a currently translated declaration is to be stored in loc or heap. The
bookkeeping system used for this purpose is quite simple. The FORTRAN
variable "LHCURR" is initialized to contain the characters "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 reserva-
tion is put out, the contents of LHCURR (denoted in our notation below by
"[LHCURR]") is inserted in the translated program.
Array Declarations

There are two parts to the problem of generating arrays in a compiled
ALGOL 68 program. The first part involves construction of a run-time de-
scriptor 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
\( \psi_{4a1} \) records in the \( \text{OBJL3C} \) stack where translated code for the descriptor
begins, so that \( \psi_{4a2} \) can erase any code that may have been produced while
scanning the \(<\text{array generator}>\).

For the recognized program sequence

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

the compiler produces the following translated code:

```
SAVEU,
2 + (LACURR) + U,
(LACURR) + \begin{array}{c}
1 + \text{no. of} \\
\text{dimensions}
\end{array} \rightarrow \text{LACURR},
0 \rightarrow \text{[U]},
Translation of \(<\text{rows}>\),
U = \begin{array}{c}
1 + \text{no. of} \\
\text{dimensions}
\end{array} \rightarrow \text{TEMP1},
0 \rightarrow \text{[TEMP1]},
[U](36 + 53) \rightarrow \text{SIZE},
SIZE > 0: U \rightarrow \text{[TEMP1]}(0 \rightarrow 17),
(LACURR) + \text{SIZE} \rightarrow \text{LACURR};
no. of \begin{array}{c}
dimensions
\end{array} \rightarrow \text{[TEMP1]}(36 \rightarrow 53),
P\#FU,
```
The run-time variable \( U \) in the above code is used by the compiler sub-

routines as an index for placing information in the current word of the

array descriptor being generated. As can be seen in Figure 2, the de-
scriptor consists of \( n+1 \) words, with \( n \) the number of dimensions. Thus,

the first descriptor word storing bounds is located at address

"2 + \{LICURR\}" above (see Preliminaries for meaning of \{LICURR\}), and

that word is set to zero by the "0 + \{U\}" statement.

The statement that reserves space for the descriptor in \{LICURR\} is

inserted later by subroutine \( \psi_{\text{map}} \) after the number of dimensions is known

by the compiler. Translation of \(<\text{rows}>\) 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:

\[
\text{SIZE} > 0: \quad U + [\text{TEMP}](0 + 17),\{LICURR\} + \text{SIZE} + \{LICURR\};
\]

Figure 2 below displays the storage structure used for run-time array

descriptors. The "\( a_1u_1 \)" notation represents a two-bit code for indicating

whether or not the corresponding lower and upper bounds can be changed

dynamically. When \( a_1 \) (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 \(<\text{rows}>\).
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>$i_1u_1$</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>$i_nu_n$</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:

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

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

Subroutine $\psi_{7a}$ is called after the translation of the first pair of bounds in an array declaration, and subroutine $\psi_{7b}$ is called for each subsequent pair of bounds in the declaration. Primarily, $\psi_{7a}$ and $\psi_{7b}$ work together to count dimensions and calculate the strides of the descriptor. They assume that translation rules $P_{13}, P_{12}, P_{11},$ and $P_{10}$ have inserted the upper and lower bounds of each dimension into the appropriate fields of the descriptor word. The code put out by $\psi_{7b}$ gives an idea of this interaction:
[U](18 + 35) - [U](0 + 17) → SIZE,
(Subtracts the lower bound from the upper bound.)
SIZE * [U](36 + 53) + SIZE,
(Calculates the value of the next stride.)
U + 1 + U,
0 + [U],
(Zeroes the next word in the descriptor.)
SIZE > 0 : SIZE = [U](36 + 53);;
(Stores the stride in the next word of the descriptor.)

When the array descriptor is completed, there may follow code for initializing each element of the resulting run-time array. This code is written out for rules P₃b and P₃d:

P₃b: <stowed declarer> + <array generator>/φ₃b1/
    STRUCT(/φ₃b2/ <fields>/)/φ₃b3

P₃d: <stowed declarer> + <array generator>/φ₃d1/
    <mode indicant>/φ₃d2/

Essentially, subroutines φ₃d1 and φ₃d2 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 LHCURR. Since translation of P₃b works in a very similar fashion, we will give as our illustration the code produced by φ₃d1 and φ₃d2:
0 → INDEX REGISTER 2,  
U = 1 + U,  
LABEL 2:  
INDEX REGISTER 2 + 1 → INDEX REGISTER 2,  
INDEX REGISTER 2 > SIZE: LABEL 3.;;  
\( \text{[LHCURR]} + 1 + \text{[U + INDEX REGISTER 2]}, \)  
Translation of \(<\text{mode indicant}>, \)  
(This includes a jump to the definition of the \(<\text{mode indicant}> \) 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 \( \text{[LHCURR]} \) was set equal to the last address of this region by compiler subroutine \( \psi_{682}. \) Thus, the first address in any data structure created by the \(<\text{mode indicant}> \) is given by \( \text{[LHCURR]} + 1 \), and this address is the value used to initialize the corresponding array elements iteratively.

Structure Declarations  
In our implementation of ALGOL 68, the fields of data structure can store plain values or they can be linked to other structures and arrays. During a declaration, then, code linking a structure field to another structure or array may or may not be called for. On a syntactic basis, we say that such linking code is called for in a structure having some field that corresponds to a \(<\text{stowed declarer}> \) or \(<\text{mode indicant}> \). The code is not compiled in the case of a structure field that corresponds to a \(<\text{nonstowed declarer}> \). To simplify our discussion of the translation process, we consider only rule \( \psi_{15} \) of Appendix 1, since it unes...
essentially the same mechanisms as rule $P_{3b}$:

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

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

SAVE U,

\{LHCURR\} $\rightarrow$ U,

\{LHCURR\} + \text{<no. of fields>} + \{LHCURR\},

Translation of <fields>,

PMP 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 LSC 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:

$$\text{<no. of intervening nonstowed fields>} + U \rightarrow U,$$

\{LHCURR\} + 1 + \{U\},

Translation of <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 LSC 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

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

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

\[
\begin{align*}
L & \text{ABEL 5}, \\
L & \text{ABEL 4:} \\
\text{Translation of } \langle \text{actual mode declarer} \rangle, \\
[ & \text{RETURN}] \ ., \\
L & \text{ABEL 5:}
\end{align*}
\]

In the above code, "LABEL 4" is the unique program label corresponding to the definition of the \(<\text{mode indicant}>\). "LABEL 5" is used to isolate the indicant definition in the run-time program. With this isolation, the only way to execute the code produced for \(<\text{actual mode declarer}>\) is to jump to \( \text{LABEL 4} \). 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-defineational function, we need a function call in the form of a \(<\text{mode indicant}>\) appearing in some declaration. This function call produces the following code:

\[
\begin{align*}
\text{SAVE RETURN}, \\
J + 2 \rightarrow \text{RETURN}, \\
\text{LABEL 4},. \\
\text{POP RETURN}
\end{align*}
\]
For this calling sequence, LABEL 4 is used as the location of the particular <mode indictant> 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 "PP RETURN". All the compiler needs to do is carry a record of the labels that correspond to the currently active mode indictants and to retain a representation of the data structures to which these indictants correspond. As before, the value of this mode indictant function is the first free location in LSC 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

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

The Compiled Code

```
LABEL 10.,
LABEL 7:
SAVE U,
LSC + U,
LSC + 1 + LSC,
U + 1 + U,
LSC + [U],
SAVE U,
2 + LSC + U,
LSC + 2 + LSC,
0 + [U],
1 + [U](35 + 53),
1 + $\theta$ERAND REGISTER 2,
1 + [U](55 + 55),
$\theta$ERAND REGISTER 2 + [U](0 + 17),
3 + $\theta$ERAND REGISTER 2,
1 + [U](54 + 55),
$\theta$ERAND REGISTER 2 + [U](18 + 35),
1 + [U](36 + 53),
[U](18 + 35) - [U](0 + 17) \rightarrow SIZE,
SIZE + 1 \rightarrow SIZE,
U + 1 \rightarrow U,
0 + [U],
SIZE > 0: SIZE + [U](36 + 53);;
```
U - 2 + TEMP1,
O + [TEMP1],
[u](36 + 53) + SIZE,
SIZE > 0: U + [TEMP1](0 + 17),
LOC + SIZE + LOC;;
1 + [TEMP1](36 + 53),
POP 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(S8$),
POP RETURN
POP U,
LABEL 8. ,
LABEL 9:
PPOP U,
[RETURN],
LABEL 10:
Bibliography

References Cited in Text


Appendix 1
A Partial Translation Grammar for ALGOL 68

\[ P_{1a}: \langle \text{mode declarer} \rangle \rightarrow \langle \text{stowed declarer} \rangle / P_{1a} \]
\[ P_{1b}: \quad | \langle \text{nonstowed declarer} \rangle / P_{1b} \]
\[ P_{1c}: \quad | \langle \text{mode indicator} \rangle / P_{1c} \]
\[ P_{2a}: \langle \text{nonstowed declarer} \rangle \rightarrow \text{REF} / P_{2a1} \quad | \text{mode declarer} / P_{2a2} \]
\[ P_{2b}: \quad | \langle \text{mode declarer} \rangle / P_{2b} \]
\[ P_{3a}: \langle \text{stowed declarer} \rangle \rightarrow \text{STRUCT} \langle / P_{3a1} \langle \text{fields} \rangle \rangle / P_{3a2} \]
\[ P_{3b}: \quad | \langle \text{array generator} \rangle / P_{3b1} \langle \text{STRUCT} \langle / P_{3b2} \langle \text{fields} \rangle \rangle \rangle / P_{3b3} \]
\[ P_{3c}: \quad | \langle \text{array generator} \rangle \langle \text{nonstowed declarer} \rangle / P_{3c} \]
\[ P_{3d}: \quad | \langle \text{array generator} \rangle \langle / P_{3d1} \langle \text{mode indicator} \rangle \rangle / P_{3d2} \]
\[ P_{4a}: \langle \text{array generator} \rangle \rightarrow \langle [ / P_{4a1} \langle \text{rows list} \rangle ] / P_{4a2} \]
\[ P_{5a}: \langle \text{fields} \rangle \rightarrow \langle \text{field} \rangle / P_{5a1} \]
\[ P_{5b}: \quad | \langle \text{field} \rangle / P_{5b1} \langle / P_{5b2} \langle \text{fields} \rangle \rangle \]
\[ P_{6a}: \langle \text{field} \rangle \rightarrow \langle \text{field mode declarer} \rangle \langle \text{field selector} \rangle / P_{6a} \]
\[ P_{7a}: \langle \text{rows list} \rangle \rightarrow \langle \text{row} \rangle / P_{7a1} \]
\[ P_{7b}: \quad | \langle \text{row list} \rangle, \langle \text{row} \rangle / P_{7b1} \]
\[ P_{8a}: \langle \text{field mode declarer} \rangle \rightarrow \langle \text{mode declarer} \rangle \]
\[ P_{9a}: \langle \text{field selector} \rangle \rightarrow \langle \text{name} \rangle / P_{9a} \]
\[ P_{10a}: \langle \text{row} \rangle \rightarrow : / P_{10a} \]
\[ P_{10b}: \quad | \langle \text{lower bound} \rangle : \langle \text{upper bound} \rangle \]
\[ P_{11a}: \langle \text{lower bound} \rangle \rightarrow \langle \text{bound} \rangle / P_{11a} \]
\[ P_{12a}: \langle \text{upper bound} \rangle \rightarrow \langle \text{bound} \rangle / P_{12a} \]
\[ P_{13a}: \langle \text{bound} \rangle \rightarrow \langle \text{formula} \rangle / P_{13a} \]
\[ P_{13b}: \quad | \langle \text{either} \rangle / P_{13b} \]
\[ P_{13c}: \quad | \langle \text{flex} \rangle / P_{13c} \]

* This syntax has been tested, and is certified to be LR(1).
\[
P_{13} : \text{<formula EITHER/13d/>}
\]
\[
P_{13e} : \text{formula FLEX/13e/>}
\]
\[
P_{14a} : \text{<mode decarator> \rightarrow <procdecl>}
\]
\[
P_{14b} : \text{<union mode decl>}
\]
\[
P_{14c} : \text{REAL/14c/>}
\]
\[
P_{14d} : \text{INT/14d/>}
\]
\[
P_{14e} : \text{BOOL/14f/>}
\]
\[
P_{14f} : \text{CHAR/14f/>}
\]
\[
P_{15a} : \text{<procdecl> \rightarrow PROC/15a/>}
\]
\[
P_{15b} : \text{PROC/15b/<virtual procplan>/15b2/>}
\]
\[
P_{16a} : \text{<virtual procplan> + <virtual parameters>/16a/>}
\]
\[
P_{16b} : \text{<virtual mode declarer>/16b/>}
\]
\[
P_{16c} : \text{<virtual parameters> + <virtual mode declarer>/16c/>}
\]
\[
P_{17a} : \text{<virtual parameters> + <virtual mode declarer>/17a/>}
\]
\[
P_{17b} : \text{<virtual mode declarer>/17b/<virtual parameters>/17b2/>}
\]
\[
P_{18a} : \text{<union mode decl> + UNION (<18a1/非union indicants>) /18a2/>}
\]
\[
P_{19a} : \text{<nonunion indicants> + <mode indicant>/19a/>}
\]
\[
P_{19b} : \text{<mode indicant> /19b1/<nonunion indicants>/19b2/>}
\]
\[
P_{20a} : \text{<mode indicant> \rightarrow $ <name> $/20a/>}
\]
\[
P_{21a} : \text{<virtual mode declarer> \rightarrow <mode declarer>/21a/>}
\]
\[
P_{22a} : \text{<actual mode declarer> \rightarrow <mode declarer>/22a/>}
\]
\[
P_{23} : \text{<formal mode declarer> \rightarrow <mode declarer>}
\]
\[
P_{24a} : \text{<declaration> + <mode declaration>}
\]
\[
P_{24b} : \text{<operator declaration>}
\]
\[
P_{24c} : \text{<initialization>}
\]
\[
P_{24d} : \text{<priority declaration>}
\]
\[ P_{25a} : \quad \text{<mode declaration> } \rightarrow \text{MODE/}^{25a1} \text{/mode indiciant} = /^{25a2} \]

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

\[ P_{26a} : \quad \text{<operator declaration> } \rightarrow \text{OP/}^{26a1} \text{/op.} = /^{26a2} \text{/<routine>/}^{26a3} \]

\[ P_{27a} : \quad \text{<Initialization> } \rightarrow \text{<reference to actual declarer>/}^{27a} \text{/<name list> } \]

\[ P_{27b} : \quad \mid \text{<reference to mode global generator>/}^{27b} \text{/<name list> } \]

\[ P_{27c} : \quad \mid \text{<formal mode declarer> <tag> =/}^{27c1} \text{/<tertiary>/}^{27c2} \]

\[ P_{28a} : \quad \text{<name list> } \rightarrow \text{<tag>/}^{28a} \]

\[ P_{28b} : \quad | \text{<tag> } : = /^{28b1} \text{/<tertiary>/}^{28b2} \]

\[ P_{28c} : \quad | \text{<name list>}, \text{<tag>/}^{28c} \]

\[ P_{28d} : \quad | \text{<name list>}, \text{<tag> } : = /^{28d1} \text{/<tertiary>/}^{28d2} \]

\[ P_{29a} : \quad \text{<reference to actual declarer> } \rightarrow \text{<actual mode declarer>/}^{29a} \]

\[ P_{30a} : \quad \text{<reference to mode global generator> } \rightarrow \text{HEAP/}^{30a1} \text{/<actual mode declarer>/}^{30a2} \]

\[ P_{31a} : \quad \text{<program> } \rightarrow \text{ENTRY$/}^{31a1} \text{/<particular program>$EXIT$/}^{31a2} \]

\[ P_{32a} : \quad \text{<particular program> } \rightarrow \text{<closed clause} \]

\[ P_{33a} : \quad \text{<closed clause> } \rightarrow \text{BEGIN/}^{33a1} \text{/<serial clause>END/}^{33a2} \]

\[ P_{33b} : \quad | (/^{33a1} \text{/<serial clause>})/^{33a2} \]

\[ P_{33c} : \quad | (/^{33a1} \text{/<row of clause>})/^{33a2} \]

\[ P_{34a} : \quad \text{<serial clause> } \rightarrow \text{<declaration prelude sequence>;<suite of clause train> } \]

\[ P_{34b} : \quad |\text{<declaration prelude sequence;} > \text{<label completer> <suite of clause train> } \]

\[ P_{34c} : \quad |\text{<label completer> <suite of clause train> } \]

\[ P_{34d} : \quad |\text{<suite of clause train> } \]

\[ P_{35a} : \quad \text{<declaration prelude> } \rightarrow \text{<unitary clause>;} \text{<declaration prelude> } \]

\[ P_{35b} : \quad |\text{<declaration> } \]

\[ P_{36a} : \quad \text{<declaration prelude sequence>;} \rightarrow \text{<declaration prelude sequence>;} \]

\[ P_{36b} : \quad \text{<unitary clause> } \rightarrow \text{<unitary clause>;} \]

\[ P_{36c} : \quad \text{<unitary clause EXIT> } \rightarrow \text{<unitary clause>EXIT } \]

\[ P_{36d} : \quad \text{<label> } \rightarrow \text{<label> } ; \]
P_{37a} : <suite of clause train> → <unitary clause>.

P_{37b} : |<labelstat>

P_{37c} : |<labelstat><suite of clause train>

P_{37d} : |<unitary clause;><suite of clause train>

P_{38a} : <labelstat> + <unitary clause;><label:>

P_{38b} : |<unitary clause EXIT><label:>

P_{38c} : |<labelstat><label:>

P_{38d} : <label completer> → <label>

P_{38e} : |<label completer><label:>

P_{38f} : <declaration prelude sequence> → <declaration prelude>/\texttt{34b2}/

P_{38g} : |<declaration prelude sequence;><declaration prelude>

P_{39a} : <label> + <name>

P_{40a} : <unitary clause> + <tertiary>

P_{40b} : |<confrontation>

P_{40c} : |\texttt{FOR <tag} FROM <serial clause> BY <serial clause> \texttt{TØ <serial clause>}

\texttt{DO} <unitary clause>

P_{40d} : |\texttt{FROM <serial clause> BY <serial clause> \texttt{TØ <serial clause> DO}

<unitary clause>

P_{40e} : |\texttt{WHILE <serial clause> DO <unitary clause}>

P_{41a} : <confrontation> → <identity relation>

P_{41b} : |<conformity relation>

P_{41c} : |<reference to mode assignation>

P_{41d} : |<mode cast>

P_{42a} : <reference to mode assignation> → <reference to mode tertiary>

: = <unitary clause>

P_{43a} : <reference to mode tertiary> → <tertiary>

P_{44a} : <conformity relation> → <reference to mode tertiary>

<conformity relator><tertiary>
\[ \begin{align*}
P_{45a} & : \langle \text{conformity relator} \rangle \rightarrow :: \\
P_{45b} & : \quad ::= \\
P_{46a} & : \langle \text{identity relations} \rangle \rightarrow \langle \text{reference to mode tertiary} \rangle \\
& \quad \langle \text{identity relator} \rangle <\text{reference to mode tertiary}> \\
P_{47a} & : \langle \text{identity relator} \rangle \rightarrow :: \\
P_{47b} & : \quad ::= \\
P_{48a} & : \langle \text{mode cast} \rangle \rightarrow <\text{virtual mode declarker}>/\gamma_{48a1}/<\text{unitary clause}>/\gamma_{48a2}/ \\
P_{49a} & : \langle \text{void cast} \rangle \rightarrow :<\text{unitary clause}>/\gamma_{49a}/ \\
P_{50a} & : \langle \text{tertiary} \rangle \rightarrow <\text{formula} > \\
P_{51a} & : <\text{formula} > \rightarrow <\text{formula} ><\text{op} ><\text{unary} > \\
P_{51b} & : \quad <\text{unary} > \\
P_{52a} & : <\text{unary} > \rightarrow <\text{op} ><\text{unary} > \\
P_{52b} & : \quad <\text{secondary} > \\
P_{53a} & : \langle \text{priority declaration} \rangle \rightarrow \text{PRIORITY}<\text{op} >=<\text{nonzero digit} > \\
P_{54a} & : <\text{mode cast} > \rightarrow <\text{mode cast} > \\
P_{54b} & : \quad <\text{void cast} > \\
\end{align*} \]

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

\[ \begin{align*}
P_{61a} & : <\text{op} > \rightarrow * \\
P_{61b} & : \quad | - \\
P_{61c} & : \quad | * \\
P_{61d} & : \quad | / \\
P_{61e} & : \quad | / . . \\
P_{61f} & : \quad | + \\
P_{61g} & : \quad | - \\
P_{61h} & : \quad | \\
P_{61i} & : \quad | \} \\
P_{61j} & : \quad | \} 
\end{align*} \]
\( p_{61k} \) | >
\( p_{61l} \) | <
\( p_{61m} \) | ≥
\( p_{61n} \) | ≤
\( p_{61o} \) | =
\( p_{61p} \) | ≠
\( p_{61q} \) | :=
\( p_{61r} \) | +:=
\( p_{61s} \) | *:=
\( p_{61t} \) | /:=
\( p_{61u} \) | ./.:=
\( p_{61v} \) | \( \text{RE} \)
\( p_{61w} \) | \( \text{IM} \)
\( p_{61x} \) | \( \text{ODD} \)
\( p_{61y} \) | \( \text{SIGN} \)
\( p_{61z} \) | \( \text{ROUND} \)
\( p_{61aa} \) | \( +<\text{name}> \)
\( p_{62a} \) | \( \text{secondary} \) \( \rightarrow \) \( <\text{primary}> \)
\( p_{62b} \) | \{<\text{cohesion}> \}
\( p_{63a} \) | \( \text{cohesion} \) \( \rightarrow \) \( <\text{selection}> \)
\( p_{63b} \) | \{<\text{generator}> \}
\( p_{64a} \) | \( <\text{selection}> \) \( + <\text{name}> \) \( \text{OF} <\text{secondary}> \)
\( p_{65a} \) | \( <\text{generator}> \) \( \rightarrow \) \( <\text{reference to mode global generator}> \)
\( p_{65b} \) | \{<\text{reference to mode local generator}> \}
\( p_{66a} \) | \{<\text{reference to mode local generator}> \( \rightarrow \) \( <\text{actual mode declarer}> \)
\( p_{67a} \) | \( <\text{primary}> \) \( \rightarrow \) \( <\text{base}> \)
\( p_{67b} \) | \{<\text{closed clause}> \}
\( p_{67c} \) | \{<\text{conditional clause}> \}
\[ P_{68a} : \text{<base> + <tag>} \]
\[ P_{68b} : \quad \text{<denotation>} \]
\[ P_{68c} : \quad \text{<slice>} \]
\[ P_{68d} : \quad \text{<call>} \]
\[ P_{68e} : \quad \text{GO TO <label>} \]
\[ P_{68f} : \quad \text{SKIP} \]
\[ P_{68g} : \quad \text{NIL} \]
\[ P_{69a} : \quad \text{<tag> + <name>} \]
\[ P_{70a} : \quad \text{<name> + <letter>} \]
\[ P_{70b} : \quad \text{<name> <letter>} \]
\[ P_{70c} : \quad \text{<name> <digit>} \]
\[ P_{70d} : \quad \text{<name> *} \]
\[ P_{71a} : \quad \text{<denotation> + <number>} \]
\[ P_{71b} : \quad \text{<character denotation>} \]
\[ P_{71c} : \quad \text{<string denotation>} \]
\[ P_{71d} : \quad \text{<bits>} \]
\[ P_{71e} : \quad \text{<routine>} \]
\[ P_{72a} : \quad \text{<routine> \rightarrow ("<formal parameters>"<moid case>*)} \]
\[ P_{72b} : \quad \text{<moid cast pack>} \]
\[ P_{73a} : \quad \text{<moid cast pack> \rightarrow ("/73a1/<moid cast>*/)/73a2/} \]
\[ P_{74a} : \quad \text{<formal parameters> + <formal parameters>} \]
\[ P_{74b} : \quad \text{<formal parameter;> <formal parameter>;} \]
\[ P_{75a} : \quad \text{<formal parameter;> \rightarrow <formal parameter>;} \]
\[ P_{76a} : \quad \text{<formal parameter> + <formal mode declarer> <name>} \]
\[ P_{77a} : \quad \text{<call> \rightarrow <primary>/77a1/<actual parameters>/77a2/} \]
\[ P_{78a} : \quad \text{<actual parameters> \rightarrow <unitary clause>/78a1/} \]
\[ P_{78b} : \quad \text{<unitary clause>,/78b1/<actual parameters>/78b2/} \]
\[ P_{79a} : \quad \text{<slice> \rightarrow <primary> [-indexer]>} \]
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/asyarakat/THEN/�/ELSE/�/ELSE/�/

P_82b: IF/asyarakat/THEN/�/ELSE/�/

P_82c: CASE/asyarakat/IN/�/row of clause/ELSE/�/

P_82d: CASE/asyarakat/IN/�/row of clause/ELSE/�/

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>
\[ P_{90a} : \text{octal integer} \rightarrow \text{octal digit} \]
\[ P_{90b} : \quad \mid \text{octal integer} \rightarrow \text{octal digit} \]
\[ P_{91a} : \text{octal digit} \rightarrow 0 \]
\[ P_{91b} : \quad \mid \text{nonzero octal} \]
\[ P_{92a} : \text{nonzero octal} \rightarrow 1 \]
\[ \quad \quad \quad \quad \quad \quad \]
\[ P_{92g} : \quad \mid 7 \]
\[ P_{93a} : \text{digit} \rightarrow 0 \]
\[ P_{93b} : \quad \mid \text{nonzero digit} \]
Appendix II: TRANSLATION RULES WRITTEN IN FORTRAN

SUBROUTINE CLEAN
ENTRY PSI1A
ENTRY PSI1C
ENTRY PSI1E
ENTRY PSI2A1
ENTRY PSI2A2

CODE FOR THIS NONSTOWED DECLARER IS DESTROYED AT THIS POINT. INFORMATI
ABOIJT FORMAL DECLARERS IS PASSED ON.

IF (LINK(CONT(N(I)),1),ED,3RF RM) GO TO 101
ITEMP=3RVCT
GO TO 102
101 ITEMP=3RF RM
102 ITEMP=IFETCH(1)
CALL STRIND (4LORF,ITEMP)
CALL SETIND (N(I),ITEMP,2)
CALL SETIND (ITEMP1,ITEMP,1)
N(I)=ITEMP
Z=CODELOC(K)
CALL REDUCE (K)
RETURN
ENTRY PSI2B

CALL SETIND (3RVCT,N(I),1)
RETURN
ENTRY PSI3A1
ENTRY PSI3B2

A STORAGE RESERVAION COMMAND FOR THE STRUCTURE IS INSERTED HERE
BY SUBROUTINE PSI3A2

K=K+1
CODELOC(K)=Z+3
K=K+1
CODELOC(K)=Z+4
FIELDNO(K)=1
CALL OUTCDA2 (Z,6,HSAVEU,10H // 560
CALL OUTCDA2 (Z,1,LHCCURR,10H // 570

CODER FOR THE OUTER STRUCTURE AND ITS FIRST FIELD IS INSERTED HERE BY 580
RULE PSI3A2.  590

7=7+2  590
CALL OUTCDA2 (Z,LHCCURR,10H // 600
RETURN  610

ENTRY PSI3A2  620

ALL CODE FOR FIELDS HAS BEEN GENERATED BY THIS POINT. THE REMAINING 630
CODE FOR THE OUTER STRUCTURE IS INSERTED HERE.  640

ITEM=IFETCH(1)  650
CALL STRING (4,LIST,ITEMP)  660
CALL SETIND (M(I),ITEMP,2)  670
CALL SETIND (LINK(CONT(N(I))+1),ITEMP,1)  680
COUNTNO=13+ECON(T(N(I)),10)  690
CALL PUTIND (COUNTNO,ITEMP,10)  700
N(I)=ITEMP  710
CALL REDUCE (K)  720

CEARS THE STACK OF INFORMATION REGARDING THE FIELDS.  730

INDEX=CODELOC(K)  740
CALL OUTCDA6 (Z,LHCCURR,2H /*,CONVERT (COUNTNO,2R */),2H /*,LHCCURR,7H, 750
1 //  760
IF (COUNTNO,GT,FIELDNO(K)) G0 TO 103  770

IF EQUAL, THIS MEANS THAT NONE OF THE STRUCTURE FIELDS ARE STOWED, 780
AND HENCE, NO SUBSTRUCTURES WERE CREATED.  790

ITEM=CODELOC(K)+1  800
CALL OUTCDB1 (ITEMP,2H//)  810
CALL OUTCDB1 (ITEMP,2H//)  820
GO TO 104  830
103 CALL OUTCDA1 (7,10HP0PU, //)  840
104 CALL REDUCE (K)  850
RETURN  860

ENTRY PSI3B1  870
ENTRY PSI3C1  880
ENTRY PSI3C2  890

ITEM=CONVERT(R,2RIR)  900
R=R+1  910
IF (R,GT,MAXR) MAXR=R  920
CALL OUTCDA3 (Z,3H0/* ,ITEMP,10H, //)  930
CALL OUTCDA2 (Z,10H - 1 /* U,10H //)  940
S=S+1  950
K=K+1  960
CODELOC(K)=CONVERT(S+1,2R L)  970
K=K+1  980
THIS ROUTINE IS ALMOST PSI3A2 FOLLOWED BY PSI3D2, WITH FACILITIES FOR CARRYING OVER INFORMATION ABOUT VIRTUAL OR ACTUAL DECLARATIONS.

ITEM = IFETCH(1)
CALL STRING (4L0STR,ITEM)
CALL SETIND (N(I),ITEM,2)
CALL SETINC (LINK(CONT(N(I))+1),ITEM,1)
COUNTNO = BYTE(CONT(N(I)),10)
CALL PUTIND (COUNTNO,ITEM,10)
N(I) = ITEM
CALL PREDUCE (K)
INDEX = CODELOC(K)
CALL OUTCDB6 (INDEX,LHCCUR,5H + ,CONVERT(COUNTNO,2R ),5H $,L
1 HCURR,10H, //)
IF (COUNTNO,GT,FIELDNO(K)) GO TO 105
INDEX = CODELOC(K) - 2
CALL OUTCDB1 (INDEX,2H//)
CALL OUTCDB1 (INDEX,2H//)
GO TO 106
105 CALL OUTCDA1 (Z,10HP0PU, //)
106 CALL PREDUCE (K)
    CALL OUTCDA2 (Z,CODELOC(K),10H, //)
    CALL OUTCDA2 (Z,CODELOC(K),10H1 //)
    CALL REDUCE (K)
    CALL SETIND (N(I),N(I)-1,2)

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

ITEM = LINK(CONT(N(I)),1)
ITEM = LINKCAT(N(I)-1),1)
IF (ITEM,EQ3RACT,AND,ITEM1,EQ3RACT) GO TO 108
IF (ITEM,EQ3RFRM,OR,ITEM1,EQ3RFRM) GO TO 107
CALL SETIND (3RFRT,N(I)-1,1)
GO TO 108
107 CALL SETIND (3RFRT,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
 CHARACTER ARRAYS. THERE IS NO NEED TO ERASE ANY CODE FOR NONSTOWED DECLARERS. SINCE RULES PSI2A2, PSI44, PSI18, AND PSI19 DO THIS.

ITEMP=LINK(CONT(I),3)
IF (ITEMP.EQ.3RBIT) GO TO 110
IF (ITEMP.EQ.3RAYT) GO TO 112
109 CALL SETIND (N(I),N(I-1),2)
   I=I-1
RETURN
110 INC=10H/60*R(U-1)
111 R=R+1
   S=S+1
IF (R.GT.MAXR) MAXR=R
ITEMP=CONVTR(R,2RIR)
ITEMP1=CONVTR(S,2R L)
CALL OUTCDA3 (Z,3H0,ITEMP,10H, ) //
CALL OUTCDA3 (Z,ITEMP1,6H1TU-,ITEMP,9H1(56+53),INC,ITEMP,10H1(3
16-53), ,10H ) //
CALL OUTCDA4 (Z,ITEMP,6H +1 ,ITEMP,10H,  ) //
CALL OUTCDA5 (Z,ITEMP,10H<DIMENSION,1H1,ITEMP1,10H.+11  ) //
R=R-1
   GO TO 109
112 INC=10H12*R(U-1)
   GO TO 111
C ENTRY PSI3C2
C
CALL OUTCDA2 (Z,ICODELOC(K),10H,  ) //
K=K+1
CALL OUTCDA (Z,ICODELOC(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
ICODELOC(K)=Z
FIELDNO(K)=0
CALL OUTCDA1 (Z,10HSAVEU,  ) //
CALL OUTCDA2 (Z,LHCURR,10H+2*U ) //
Z=Z+1
CALL OUTCDA2 (Z,9H80*U1,10H ) //
C THIS OUTPUT CODE MAKES U THE INDEX OF THE DESCRIPTOR BEING CREATED.
C A SPACE IS LEFT IN THE OUTPUT CODE FOR LATER INSERTION OF A STATEMENT
C OF THE FORM LHCURR+DIMENSION+LHCURR THAT RESERVES SPACE FOR THE
C ARRAY DESCRIPTOR ON EITHER THE HEAP OR THE LOCAL STACK. RULE PSI4A2
C MAKES THIS INSERTION
C
RETURN
C ENTRY PSI4A2

CAUSES THE TOTAL SIZE OF THE ARRAY TO BE CALCULATED, SEES WHETHER THE
ARRAY IS VIRTUAL, AND STORES THE DIMENSION IN THE DESCRIPTOR.

BOUNDNO=LINK(FIELDNO(K),1)
I=I+1
N(I)=IFETCH(1)
CALL 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 (3RVRT,N(I),1)

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

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

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

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

WE NOW HAVE THE INFORMATION THAT PSI781 STORED THE SIZE OF THE ARRAY
IN U AT RUN TIME, WHERE U IS CURRENTLY THE ADDRESS OF THE WORD THAT
FOLLOWS THE DESCRIPTOR IN MEMORY.

CALL OUTCOD6 (ITEM,LHCURR,4H,*ITEM1,4H,*LHCURR,5H, //)
CALL REDUCE (K)
CALL OUTCOD4 (Z,3HU,-ITEM1,8H,*ITEM1,10H, //)
CALL OUTCOD2 (Z,10H,0*(ITEM1),10H, //)
CALL OUTCOD2 (Z,10HUI(36,53),10H+SIZE, //)
CALL OUTCOD3 (Z,7H,SIZE,0,10HUI(ITEM1),10H+17, //)
CALL OUTCOD4 (Z,LHCURR,10H+SIZE, ,LHCURR,10H11 //)
CALL OUTCOD3 (Z,ITEM1,10H-1*(ITEM1),10H(35,53),//)

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(1),EQ.6$RSTOWED) GO TO 116
Z=CODELOC(K)-1
FIELDNO(K)=1+FIELDNO(K)
GO TO 117
116 ITEM$=CODELOC(K)
CALL OUTFDR2 (ITEM$CONV(FIELDNO(K),2R ),10H+U*U, //
CALL OUTFDR2 (ITEM$,LHCUUR,10H+U, //
FIELDNO(K)=1
117 CODELOC(K)=Z+1
RETURN
ENTRY PSIS$1

LINKS TOGETHER TWO FIELDS OF A STRUCTURE AT COMPIL$E TIME. COUNTS THE
NUMBER OF FIELDS, COMBINES INFORMATION ABOUT ACTUAL AND VIRTUAL FIELDS

CALL SETIND (N(I),N(I-1),2)
CALL PUTIND (1+BYTE(CONT(N(I))),10H+U, //
ITEM$=LINK(CONT(N(I)-1),1)
FIELDNO=LINK(CONT(N(I)-1),1)
IF (ITEM$,EQ.3RACT,AND,ITEM$,EQ.3RACT) GO TO 120
IF (ITEM$,EQ.3RFRM.OR,ITEM$,EQ.3RFRM) GO TO 118
IF (ITEM$,EQ.3RVC$T.OR,ITEM$,EQ.3RVC$T) GO TO 119
CALL SETIND (3RVC$T,N(I-1),1)
GO TO 120
118 CALL SETIND (3RFRM,N(I-1),1)
GO TO 120
119 CALL SETIND (3RFRM,N(I-1),1)
120 I=I+1
RETURN
ENTRY PSIT$A1

CALL SETIND (1,FIELDNO(K),1)
CALL OUTFDR1 (Z=10H+U*U),10H53) //
CALL OUTFDR1 (Z=10H+U),18-35,10H-1*U(0-17),6H$SIZE,10H+U*U, //
1)
CALL OUTFDR4 (Z=U,7H$SIZE>0,10H$SIZE$EU),3,10H6$53) //
RETURN
ENTRY PSIT$B1

C COUNTS UP THE NUMBER OF DIMENSIONS, STORES THE STRIDE IN THE CURRENT
C DESCRIPTOR WORD, AND ZEROES THE NEXT DESCRIPTOR WORD.
C
CALL SETIND (+LINK(FIELDNO(K),1),FIELDNO(K),1)
CALL OUTFDR1 (Z=10H+U18-35,10H-1*U(0-17),10H$SIZE, //
C
C CALCULATES UPPERBOUND-LOWERBOUND.
CALL OUTDA3 (Z,10H$SIZE*U1(3,10H6*53),SIZE,10H,  //)
CALL OUTDA1 (Z,10H0+1=U,  //)
7=2+1
CALL OUTDA4 (Z,6H0*(U),7H$SIZE=01,10H$SIZE={U}(3,10H6*53))); //)
RETURN

ENTRY P518A

GENERATES A CELL IN THE COMPILE-TIME LIST COPY OF A STRUCTURE
DECLARATION.

ITEMP=IFETCH(2)
CALL STRING (N(I),ITEMP+1)
I=I-1
IF (TYPE(I),EQ,6KSTONEO) GO TO 121
CALL STRING (4L0+FL,ITEMP)
GO TO 122
121 CALL STRING (4L1+FL,ITEMP)
122 CALL SETING (N(I),ITEMP,2)
N(I)=ITEMP
RETURN

ENTRY P519A

I=I+1
N(I)=NAME
RETURN

ENTRY P5110A

THIS ROUTINE COUNTS VIRTUAL BOUNDS IN AN ARRAY DECLARATION.

CALL SETDIR (1+LINK(FIELDNO(K),2),FIELDNO(K),2)

; NOTE THAT ANYTHING BUT ZERO IN THE SECOND FIELD OF FIELDNO INDICATES
; THAT THE ARRAY IS NOT AN ACTUAL ARRAY.

RETURN

ENTRY P5111A

ITEMP1=10H$U1(55*55)
ITEMP2=10H$U1(0*17)
GO TO 123

ENTRY P512A

ITEMP1=10H$U1(54*54)
ITEMP2=10H$U1(18*35)
123 GO TO (124,126,128,125,125), BOUND
124 CALL OUTDA3 (Z,5H01*,ITEMP1,10H,  //)
125 CALL OUTDA4 (?,CONVERT(P,2R T),1M*,ITEMP2,10H,  //)
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 PSI13D
    SOUND=4
    GO TO 127
    ENTRY PSI13E
    SOUND=5
127 IF (SAMEYF(N(I),INTEGER,REF)) GO TO 128
    CALL ERROR(1100NON-INTEGER ARRAY BOUND IN ROUNOSLIST.
    RETURN
128 CALL OUTEVAL (P,REF,OUTCODE,Z,1000-2)
    RETURN
    ENTRY PSI14C
    ITEMP=3RREA
    GO TO 130
    ENTRY PSI14D
    ITEMP=3RINT
    GO TO 130
    ENTRY PSI14E
    ITEMP=3RRAIT
    GO TO 130
    ENTRY PSI14F
    ITEMP=3RBYT
    GO TO 130
    ENTRY PSI15A
129 ITEMP=3OPRC
130 I=I+1
    ITEM=IFETCH(1)
    CALL SETIND (ITEM,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 ITEM=IFETCH(1)
CALL SETIND (N(I),ITEMP,ITEMP1)
I=I-1
CALL SETINC (ITEM,N(I),2)
RETURN
ENTRY PSI16C

ITEM1=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 ITEM=IFETCH(1)
CALL SETIND (3PRM,ITEMP,3)
CALL SETINC (N(I),ITEMP,2)
CALL PUTINC (ITEMP1,ITEMP,10)
N(I)=ITEMP
RETURN
ENTRY PSI1782

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

CALL PUTINO (1+BYTECONT(N(I)),101,N(I-1),10)
I=I-1
RETURN
ENTRY PSI18A1
K=K+1
CODELOC(K)=Z
RETURN
ENTRY PSI18A2
7=CODELOC(K)
CALL REDUCE (K)
RETURN
ENTRY PSI19A
ENTRY PSI19B1
ITEMP=IFETCH(1)
CALL SETINO (3RUNI,ITEMP,3)
CALL SETINO (N(I),ITEMP,2)
N[I]=ITEMP
RETURN
ENTRY PSI19B2
CALL SETINO (N(I),N(I-1),1)
I=I-1
RETURN
ENTRY PSI20A
I=I+1
ITEMP=IFETCH(2)
N[2]=ITEMP
CALL STRING (NAME,ITEMP+1)
CALL SETINO (3,ITEMP,3)
NAME=ALIAS(NAME)

THE ALIAS FUNCTION LOOKS UP THE RUN-TIME PROGRAM LABEL ASSIGNED TO THE
MODE INDICANT, AND ADDS THE INDICANT TO THE COMPILER DECLARATION TABLE
IF NECESSARY.
CALL 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=LINKCONT(N(I),II) 
IF (ITEM.EQ.3RVRT.OR.ITEM.EQ.3RVCT) RETURN 
CALL ERROR(110HAN ATTEMPT WAS MADE TO USE A FORMAL OR ACTUAL O 
1DECLARER IN PLACE OF A VIRTUAL DECLARER. 
) 
RETURN 
ENTRY PSI22A 

A CHECK IS MADE TO SEE WHETHER THE DECLARER IN QUESTION CAN BE USED AS 
AN ACTUAL DECLARER. NO CHECK IS NEEDED FOR FORMAL DECLARERS. 
ITEM=LINKCONT(N(I),II) 
IF (ITEM.EQ.3RACONT.OR.ITEM.EQ.3RVCT) RETURN 
CALL ERROR(110HAN ATTEMPT WAS MADE TO USE A FORMAL OR VIRTUAL 
1DECLARER IN PLACE OF AN ACTUAL DECLARER. 
) 
RETURN 
ENTRY PSI26A1 
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 OUTCOB (7,NAME,10H1,/) 

THE MODE INDICANT WAS DECLARED BY PSI20A, AND NAME CONTAINS THE RUN- 
TIME LABEL ASSOCIATED WITH THE INDICANT. 
FIELDNO(K)=MODE 

MODE IS THE INDEX OF THE COMPILER MODES TABLE. 
RETURN
ENTRY PSI25A3

CALL OUTCDA3 (Z,9HRETURN=RJ,8H(30=47),,10H,R,, //)
S=S+1
ITEMP=CONVERT(S,2R L)
ITEMP=CODELOC(K)+1
CALL OUTCDA2 (ITEMP,ITEMP,10H,, //)
CALL OUTCDA2 (Z,ITEMP,10H1, //)
ITEMP=FIELDNO(K)
MODES(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 OUTCDA2 (Z,NAME,10H1, //)
FIELDNO(K)=OP
RETURN

ENTRY PSI26A3

CALL OUTCDA3 (Z,9HRETURN=RJ,8H(30=47),,10HR,R,, //)
S=S+1
ITEMP=CONVERT(S,2R L)
ITEMP=CODELOC(K)+1
CALL OUTCDA2 (ITEMP,ITEMP,10H,, //)
FIELDNO(K)
OPS(ITEMP,3)=N(I)
CALL OUTCDA2 (Z,ITEMP,10H1, //)
I=I-1
CALL REDUCE (K)
RETURN

ENTRY PSI27A

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

S=S+1
ITEMP=CONVERT(S,2R L)
FIELDNO(K)=ITEMP
ITEMP1=CODELOC(K)+1
CALL OUTCODP (ITEMP1,ITEMP,4*1,1*,LHCURR,1H*,LHCURR,8H*,  //
ITEMP1=ITEMP1-2
S=S+1
ITEMP=CONVERTS(2R L)
133 CALL OUTCODB (ITEMP1,ITEMP,10H*,  //
CALL OUTCODA3 (2,10HRETURN, RJ,10H(30*47), ,10HRJ*,  //
CALL OUTCODA2 (2,ITEMP,10H*)
ITEMP=IFETCH(1)
CALL SETIND (3REF,ITEMP,3)
CALL SETIND (N(I)),ITEMP,2)
ITEMP=ITEMP
RETURN
ENTRY PSI278
LIKE PSI274, EXCEPT THAT THE EXTRA LEVEL OF PUN-TIME INDIRECTNESS IS
SUPPLIED BY PSI3DA1.
S=S+1
ITEMP=CONVERTS(2R L)
FIELDNO(K)=ITEMP
ITEMP1=CODELOC(K)+1
CALL OUTCODP (ITEMP1,ITEMP,10H*)
S=S+1
ITEMP=ITEMP1-1
ITEMP=CONVERTS(2R L)
GO TO 133
ENTRY PSI27C1
THE TAG RULE PSI68A HAS THE EFFECT OF ENTERING THE NAME ONTO THE NAMES
TABLE, AND LEAVES THE PUN-TIME ALIAS OF <TAG> IN THE VARIABLE NAME.
CODE FOR THE VIRTUAL DECLARER IS DESTROYED
OPERAND(I)=NAME
TYPE(I)=9RNONSTORED
Z=CODELOC(K)
RETURN
ENTRY PSI27C2
IF (SAMEITYP(N(I-1),N(I),REF),AND,REF,EQ,0) GO TO 134
CALL ERROR(110HATTEMPT TO INITIALIZE A DECLARED VARIABLE
USING A <TERTIARY> OF A DIFFERENT MODE.
RETURN
134 IF (TYPE(I),NE.,9RNONSTORED) GO TO 135
P=P+1
ITEMP=CONVERT(P,2R T)
CALL OUTCODA4 (Z,OPERAND(I),1H*,ITEMP,10H*,  //
OPERAND(I)=ITEMP
135 I=I-1
OPERAND(I)=IHASHB(OPERAND(I),N(I))

SETS THE TYPE OF THE NAME AND THE REGION WHERE ITS VALUE IS STORED.
	CALL OUTCOA4(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 OUTCOA4(Z,LCURR,4H*1N,NAME,18H), //)
	CALL OUTCOA3(Z,10HSAVE RETUR,10HN,J+2=RETU,10HN, //)
	CALL OUTCOA2(Z,FIELDNO(K),10H*, //)
	CALL OUTCOA2(Z,10HPOP 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 COMARED. THE <TERTIARY> IS REFERENCED AND DEPROCEDURED TO ONE LEVEL OF REFERENCE BELOW N(I-1)

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

IF (SAMEYTP(N(I)),LINK(CONT(N(I-1),1),REF)) GO TO 159
CALL ERROR (110THE <TERTIARY> ASSIGNED TO THE <TAG> OF THIS MODE
170DECLARATION DOES NOT POSSESS THE SAME MODE AS ITS ASSIGNEE.)

RETURN
159 IF (TYPE(I),NE.9RNONSTORED) GO TO 170
P=P+1
ITEM=CONVERT(P,2R T)
	CALL OUTCOA4(Z,OPERAND(I),1H*,ITEM,10H, //)
170 CALL OUTIVAL(P,REF,OUTCODE,7,1000-Z)
I=I+1
	CALL OUTCOA4(Z,ITEM,1H*,OPERAND(I),10H, //)
	RETURN

ENTRY PSI29A1

ITEM=IFETCH(1)
CALL SETINO (39REF,ITEM,3)
CALL SETINO (N(I),ITEM,2)
N(I)=ITEM

RETURN

ENTRY PSI30A1
ENTRY PSI30A2
L4CURR=4HHEAP
HERE WE PUT OUR CODE FOR GENERATING AN EXTRA LEVEL OF REFERENCING AT
RUN TIME. THE BEGINNING IS SAVED FOR USE AS DESTINATION IN ASSIGNMENTS

CALL OUTCOA5 (Z,10HHEAP+1+HEA,10HP,HEAP+1 * ,10H!HEAP), 8HSAVEU, 
10H!HEAP+U, //)
RETURN
ENTRY PSI31A2
L4CURR=3HLOC
CALL OUTCOA3 (Z,10H U=GENERAT,10HOR VALUE, ,10HP POP U, //)

THE VALUE OF THE GENERATOR IS PRESERVED FOR USE IN ASSIGNMENTS9

OUTCODE(17,3)=10HPPOP U, //
RETURN
ENTRY PSI31A1

CALL INITIAL(NAME,MODE)
RETURN
ENTRY PSI31A2

CALL FINAL(NAME,MODE)
RETURN
ENTRY PSI33A1
ENTRY PSI33B1
L4CURR=3HLOC
CALL BLOK4GN(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
RETURN
ENTRY PSI33A2
ENTRY PSI33B2

CALL BLOKEND(P,MODE)
RETURN
PSI349 CAUSES LOCAL BLOCK STORAGE TO BE ERASED IN THE TRANSLATED
PROGRAM.

ENTRY PSI34A

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

ITEM=CODELOC(K)+2
ITEM1=ITEM+3
136 OUTCODE(ITEM1,1)=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
"ALL OUTCP2 (ITEM,10H,1*10LOKNUM*,10HLOKNUM, )
CALL OUTCP3 (ITEM,4HLOC*,10HSTORAGE[1L,10HOKNUM], )
CALL OUTCP6 (Z,10HSTORAGE[1L,10HOKNUM]+LOC,1H,10HLOKNUM-1,10H
10KNUM*,10H
RETURN

ENTRY PSI36A

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

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

CASE/PSI33A1/<SER. CL.>IN/PSI82D2/<ROW OF CL.>FI/PSI82G3/
CASE/PSI33A1/<SER. CL.>IN/PSI82D2/<ROW OF CL.>ELSE/PSI82G3/<SER. CL.>FI

ENTRY PSI82D2

IF(.NOT.SAMEYP(N(I),INTREAL,REF))
1 CALL ERROR(1IOHINDEX OF A CASE STATEMENT IS NOT A NUMERICAL QU
ANTITY,)
IF (.NOT. (TYPE(I) EQ. 9RNONSTORED)) GO TO 183
P=P+1
ITEM=CONVET(P,2R T)
CALL OUTCDA4 (Z, OPERAND(I), 1H, , TEMP, 10H,  // 8980
CALL OUTEVAL (P, REF, OUTCODE, Z, 1000-Z)  8990
I = I + 1  9000
CODELOC(K) = 7 + 1  9010
CALL OUTCDA4 (Z, TEMP, 6H 10H,  // 9020
   Z, TEMP, 6H 10H,  // 9030
CALL OUTCDA4 (Z, TEMP, 6H 10H,  // 9040
   Z, TEMP, 10H + RJ(30+47), 10H, RJ,  // 9050
   T(P) < 0 LABEL ELSE:  // 9060
   T(P) > 0 LABEL FI = 1 T(P), (SEE PSI8203 FOR COMPLETION.)  9070
   T(P) = RJ(30+47), RJ,  9080
CAUSES A JUMP TO A TABLE LEADING TO THE APPROPRIATE CLAUSE OF THE 9090
CASE STATEMENT, NOTE THAT STORAGE (BLOKNUM) IS USED TO RESET P TO THE 9100
SAME LEVEL FOR EACH CLAUSE.  9110
THE SAVING OF CODELOC(K) IN WHAT FOLLOWS IS PERFORMED FOR COMPATIBILITY 9120
WITH PSI33A1 WHEN A <ROW OF CLAUSES> APPEARS WITHIN A <COLLATERAL CLAUSE>. 9130
   CALL BLOKEND(P, MODE)  9140
   K = K + 1  9150
   CODELOC(K) = Z  9160
   RETURN  9170
ENTRY PSI8203.  9180
SEARCH HERE FOR THE MOST DEREFERENCED AND DEPROCEDURED (I.E., LOWEST) 9190
CLAUSE IN THE <ROW OF CLAUSE>, AND BRING THE REMAINING CLAUSES DOWN 9200
TO THAT LEVEL.  9210
   CALL BLOKEND(P, MODE)  9220
   CALL BLOKEND(P, MODE)  9230
   K = K + 1  9240
   S = S - 1  9250
   Z = Z + 1  9260
   LINK1 = MINIMUM (N(I))  9270
   CALL BLOKEND(P, MODE)  9280
RULES PSI83A AND PSI830 FOR <ROW OF CLAUSE> CREATE A LIST OF LABELS 9290
ON TOP OF THE OPERAND STACK, EACH ELT. OF THE LIST CONTAINS IN SEQUENCE 9300
(1) A POINTER TO THE NEXT ELT.,  9310
(2) THE LABEL WHERE THE UNITARY CLAUSE BEGINS,  9320
(3) THE LABEL WHERE DEREFERENCING AND DEPROCEDURING  9330
   OF THAT CLAUSE OCCURS,  9340
   (4) THE LOCATION IN THE CODE OF (3) WHERE ESCAPE IS  9350
   MADE TO THE CODE DIRECTLY FOLLOWING THE CASE  9360
   STATEMENT, (SEE BALANCE SUBROUTINE.)  9370
   CALL BALANCE (LINK1, ITEM1, ITEM2, S, P, Z, OUTCODE)  9380
   N(I) = LINK1  9390
   ITEM1 = OPERAND(I)  9400
   ITEM2 = N(I)  9410

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 OUTCDA2 (Z,ITEM,10H1,10H1,ITEM=CODELOC(K))
K=K-1
OUTCODE(ITEM+2,3)=ITEM

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

ITEM2=OPERAND(I)
139 CALL OUTCDA2 (Z,CONT(ITEM2+2),10H1,10H1)
ITEM2=LINK(CONT(ITEM2),1)
IF (ITEM2,GT,0) GO TO 139

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

S=S+1
ITEM=CONVERT(S,2R L)
CALL OUTCDA2 (Z,ITEM,10H1,10H1)
OUTCODE(ITEM1,3)=ITEM
OUTCODE(ITEM1+1,3)=CONVERT(INYTE(CONT(OPERAND(I)),10),2R)
OUTCODE(ITEM1+1,5)=ITEM
GO TO 184

ENTRY PSIA204

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

CALL ROKEND(P,MODE)
O=P+1
ITEM4=CONVERT(P,2R T)
IF (OPERAND(I),EQ,IITEM4) GO TO 140
CALL OUTCDA4 (Z,OPERAND(I),10H),ITEM4,10H,

140 K=K-1

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

IF (.NOT.SAMEFYP(N(I-1),N(I),REF)) GO TO 141
IF (REF,EQ,0) GO TO 142
GO TO 144
141 IF (SAMETYPE(F(N1),N(I-1),REF)) GO TO 145
CALL ERROR(110) THE REMAINING CLAUSE AFTER ELSE* IN THE CASE STATEMENT DOES NOT HAVE THE SAME NAME AS PRECEDING CLAUSES.
RETURN

10000
10010
10020
10030
10040
10050
10060
10070
10080
10090
10100
10110
10120
10130
10140
NO BALANCING IS NEEDED HERE.

142 S=S+1
ITEMP=CONVERT(S,2R L)
CALL OUTDCA2 (Z,ITEMP,10H1 //)
CALL LOOP82D (OPERAND(I-1),ITEMP,OUTCODE)

PROVIDES ESCAPE LABELS TO THE CASE CLAUSES.

143 I=I-1
TYPE(I)=68STORED
OPERAND(I)=ITEMP
RETURN

THE ENTIRE <ROW OF CL.> AFTER BEING BALANCED INTERNALLY, IS HERB BALAN THE SERIAL ELSE CLAUSE.

144 S=S+1
ITEMP=CONVERT(S,2R L)
CALL OUTDCA2 (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 CODE. NOTE THAT VALUE RETURN IS USED TO CARRY THE VALUES OF EACH CLA

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

CALL OUTEVAL (P,REF,OUTCODE,Z,1000-Z)
CALL OUTDCA2 (Z,TYPE(I),10H1 //)
M(I-1)=N(I)
GO TO 143

THE <SERIAL CL.> MUST BE DEREFERENCED AND DEPROCEDURED TO BALANCE WITH <ROW OF CL.>.

145 S=S+1
ITEMP=CONVERT(S,2R L)
CALL OUTDCA2 (Z,ITEMP,10H1 //)
CALL OUTEVAL (P,REF,OUTCODE,Z,1000-Z)
CALL LOOP82D (OPERAND(I-1),ITEMP,OUTCODE)
CALL OUTDCA2 (Z,ITEMP,10H1 //)
GO TO 143

ENTRY PSB82C3
CALL BLOKEND(P,MODE)
K=K-2
LINK1=MINIMUM(N(I))
S=S-1
Z=Z-1
CALL BALANCE (LINK1,OPERAND(I),N(I),S,P,Z,OUTCODE)
N(I)=LINK1
TYPE(I) = 6 STORED
S = S + 1
ITEMP = CONVERT(S, 2 R L)
CALL OUTCDA2 (Z, ITEMPI, 10H, //)
ITEMP = CODELOC(K)
K = K + 1
OUTCODE(ITEMP + 2, 3) = ITEMP
ITEMP = OPERAND(I)
ITEMP = ITEMP + 2
CALL OUTCDA2 (Z, CONT(ITEMP + 2), 10H, //)
ITEMP = LINK(CONT(ITEMP + 3), 1)
IF (ITEMP Cait. 0) GO TO 146
S = S + 1
ITEMP = CONVERT(S, 2 R L)
CALL OUTCDA2 (Z, ITEMPI, 10H, //)
OUTCODE(ITEMP, 3) = ITEMP
ITEMP = ITEMP + 1
OUTCODE(ITEMP, 3) = CONVRT(ITYTE(CONT(OPERAND(I), 10), 2 R)
OUTCODE(ITEMP, 5) = ITEMP
P = P + 1
ITEMP = CONVERT(P, 2 R T)
CALL OUTCDA3 (Z, SHO, ITEMPI, 10H, //)
S = S + 1
ITEMP = CONVERT(S, 2 R L)
CALL OUTCDA2 (Z, ITEMPI, 10H, //)
CALL LOOP82D (OPERAND(I), ITEMP, OUTCODE)
TYPE(M) = 6 STORED
OPERAND(I) = ITEMP
RETURN
IF /PSI82A1/ < SERIAL CL > THEN /PSI82A2/ < SERIAL CL > ELSE /PSI82A3/ < SERIAL CL > FI
IF /PSI82A1/ < SERIAL CL > THEN /PSI82A2/ < SERIAL CL > FI /PSI82B3/;
ENTRY PSI82A2
IF (.NOT. SAMEYPE(N(I), 9001, REF))
1 CALL ERROR(110, IF STATEMENT WITH NON-BOOLEAN TEST.
2 )
IF (.NOT. (TYPE(I), EQ. 9A41, STORED)) GO TO 147
P = P + 1
ITEMP = CONVERT(P, 2 R T)
147 CALL OUTCDA4 (Z, OPERAND(I), 1H, ITEMPI, 10H, //)
I = I + 1
CALL OUTCDA4 (Z, CONVRT(P, 2 R T), 3H = 01, 0, 10H, //)
CODELOC(K) = Z
CODELOC CARRIES LOCATION TO INSERT FIXUP LABEL HERE.
148 CALL BLOKEND (P, MODE)
   CALL BLOKBGN (P, MODE)
ITEMP = Z + 3
K = K + 1
CODELOC(K) = ITEMP
K = K + 1
CODELOC(K) = ITEMP
Z = ITEMPI + 1
ENTRY PSI82A3
  IF (OPFRAND(I).EQ.ITEM) GO TO 149
  ITEM=CONVERT(IBYTE(STORAGE(BLOKNUMM),10I+1,2R T)
  CALL OUTCDA4 (2,OPFRAND(I),1H+,ITEM,10H, //)
  TYPE(I)=6RSTOR0
  OPERAND(I)=ITEM
149 CALL OUTCDA2 (?0,10H+, //)
  INSERT SKIP CLAUSE LABEL HERE.

K=K-1
ITEM1=CODELOC(K)
CODELOC(K)=7
S=S+1
ITEM=CONVERT(S,2R L)
CALL OUTCDA2 (?ITEM,10H+ //)
OUTCODE(ITEM1,3)=ITEM
GO TO 148

ENTRY PSI82A4
  CALL BLOKEND(P,MODE)
P=P+1
ITEM=CONVERT(P,2R T)
IF (OPFRAND(I).EQ.ITEM) GO TO 150
CALL OUTCDA4 (2,OPFRAND(I),1H+,ITEM,10H, //)
150 K=K-1

BALANCING OF CONSEQUENCE AND ALTERNATIVE

IF (.NOT.SAMETYP(N(I-1),N(I),REF)) GO TO 151
IF (REF,EQ.0) GO TO 152
GO TO 153
151 IF (SAMESYP(N(I),N(I-1),REF)) GO TO 155
  CALL ERROR(110THE CONSEQUENCE AND ALTERNATIVE OF AN IF STATEMENT
  WOULD NOT HAVE THE SAME A POSTERIORI MODE).

NO BALANCING NEEDED.

152 S=S+1
ITEM1=CONVERT(S,2R L)
CALL OUTCDA2 (2,ITEM1,10H+ //)
ITEM2=CODELOC(K)
OUTCODE(ITEM2,1)=ITEM1
K=K-1
GO TO 154

LOWER THE CONSEQUENCE TO THE LEVEL OF THE ALTERNATIVE, AND CAUSE THE A
FIVE TO SKIP OVER THE LOWERING CODE.

153 N(I-1)=N(I)
CALL OUTCDA2 (?0,10H+, //)
ITEM=7
S=S+1

RETURN
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

154

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(ITEM1,ID)+1
ITEM1=CONVERT(P,2R T)
CALL BLOKEND (0,0)
CALL BLOKGN (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(ITEM2,3)=ITEM2
CALL OUTCDA3 (7,1H0 ,ITEM1,10H,  //
S=S+1
ITEM2=CONVERT(S,2R L)
OUTCODE(7-2,1)=ITEM2
CALL OUTCDA2 (Z,ITEM2,10H)  //
RETURN
ENTRY PSI84A

C Test to SFE whether we are compiling the first clause in a row of clause
C then, if true, insert a jump around the row of clause to where the eval
JUMP TABLE BEGINS.

P=BYTE(STORAGE(BLOKNUM),10)
ITEM=CONVERT(P+1,2R T)
IF (OPERAND(I).EQ.ITEMP) GO TO 157
CALL OUTCAD2 (2,OPERAND(I),1H,ITEMP,10H, //)
157 CALL OUTCAD3 (2,10HRETURN= RJ,8H(30*47),10HRJ, //)
ITEMP=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
ITEMP3=CONVERT(S,2R L)

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

ITEMP2=CODELOC(K-2)
CALL OUTCAD2 (ITEMP2,ITEMP3,10H1 //)
CODELOC(K-1)=ITEMP3
CODELOC(K-2)=ITEMP2-1
158 CALL STPIND (CODELOC(K-1),ITEMP1+1)
OPERAND(I)=ITEMP1

STORED ON OPERAND(I) IN A LIST CELL.

S=S+1
ITEMP3=CONVERT(S,2R L)
CODELOC(K)=ITEMP3
CALL OUTCAD2 (2,ITEMP3,10H1 //)
RETURN

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

ENTRY PSI34A

DO PSI34A WORK FIRST, THEN PIECE TOGETHER THE COMPIL- TIME REPRESENTAT
OF <ROW OF CLAUSE>.

P=BYTE(STORAGE(BLOKNUM),10)
ITEM=CONVERT(P+1,2R T)
IF (OPERAND(I).EQ.ITEMP) GO TO 159
CALL OUTCAD2 (2,OPERAND(I),1H,ITEMP,10H, //)
159 CALL OUTCAD3 (2,10HRETURN= RJ,8H(30*47),10HRJ, //)
ITEMP=IFETCH(4)
CALL STRIND (CODELOC(K-1),ITEMP+1)

C CONSTRUCT LIST OF <ROW OF CL.> PROCEDURES.
C CALL STRIND (ITEMP,OPERAND(I-1))
C
C CONSTRUCT A COMPIL- TIME STRUCTURE SKELETON REPRESENTATION OF THE DATA
ITEMP=IFETCH(1)
CALL SETINU (N(I),ITEMP,2)
I=I-1
ITEMP=IFETCH(1)
CALL SETIND (N(I),ITEMP,1)
ITEMP1=ITEMP
CALL SETINE (ITEMP,ITEMP1,1)
CALL PUTINC (2,ITEMP1,10)
N(I)=ITEMP1
RETURN
ENTRY PSI83D

ITEMP=IFETCH(1)
CALL SETINU (N(I),ITEMP,1)
I=I-1
CALL SETIND (N(I),ITEMP,2)
N(I)=ITEMP
CALL PUTINC (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
ITEMP1=ITEMP+1
DO 160 I=ITEMP,ITEMP1
160 OUTCODE(I,1)=2H//
CALL OUTCODE (CODELOC(K),CODELOC(K+1),10H, , )
GO TO 164
ENTRY PSI14802

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

ITEMP1=CONT(N(I))
ITEMP2=CONT(N(I-1))
P=P+1
ITEMP3=BYTE(ITEMP1,10)
IF (ITEMP3.EQ.1BYTE(ITEMP2,10)) GO TO 161
CALL ERROR (110HTHE <COLLATERAL CLAUSE> AND ITS A POSTERIORI MODEAR
IS NOT THE SAME LENGTH.
RETURN
161 ITEMP4=LINK(ITEMP1,3)
IF (ITEMP4.EQ.3RARY) GO TO 162
IF (ITEMP4.EQ.3RSTR) GO TO 164
CALL ERROR (110H THE A POSTERIORI MODE OF A <MOID 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, THSAVEU, LMCURR, 10H, 1, T, SAVEU, 10H,  //)

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

ITEMP4=CONVERT (ITEMP3+2, 2R)
CALL OUTCDA6 (2, LMCURR, 1M+, ITEMP4, 1M+, ITEMP4, 1M+, LMCURR, 10H,  //)
CALL OUTCDA3 (2, 10H0100000000, 10H0000, 1U1, 10H,  //)
CALL OUTCDA3 (2, 10H0200000000, 1U1, 10H0000, 1U1, 10H,  //)
CALL OUTCDA3 (2, 10H0300001000, 10H0000000000000001, 10H, 1U1, 10H,  //)
CALL OUTCDA (2, CONVERT (ITEMP3, 2R), 10H01001100, 18#35, 8H),  //)

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

ITEMP4=MATRIX (X(I))
ITEMP5=IFETCH (1)
CALL SETIND (ITEMP5, ITEMP5, 1)
CALL SETIND (ITEMP4, 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 (2, 10HPOP, U, 10H, 10H, PPOPU,  //)
I=I-1
OPERA(1)=ITEMP3
RETURN

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

154 CALL OUTCDA4 (7, THSAVEU, LMCURR, 10H+1, 1M, T, SAVEU, 10H,  //)
CALL OUTCDA6 (7, LMCURR, 1M+, CONVERT (ITEMP3, 2R), 1M+, LMCURR, 10H,  //)
10H 1
CALL LOWER (ITEMP3, LINK (ITEMP2, 2), OPERAND(I), Z, P)
GO TO 163

ENTRY PSI49A1

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

N(I)=0
OPERA(1)=0
RETURN

ENTRY PSI48A2

P = P + 1
ITEMP=CONVERT (P, 2R, T)

13430
13440
13450
13460
13470
13480
13490
13500
13510
13520
13530
13540
13550
13560
13570
13580
13590
13600
13610
13620
13630
13640
13650
13660
13670
13680
13690
13700
13710
13720
13730
13740
13750
13760
13770
13780
13790
13800
13810
13820
13830
13840
13850
13860
13870
13880
13890
13900
13910
13920
13930
13940
13950
13960
13970
IF (.NOT.(ITEMP.EQ.OPERAND(I))) GO TO 165
   CALL OUTCDX (Z,OPERAND(I),1M*,ITEMP,10,H, )
   CALL ERROR (110H THE A PRIORI MODE OF THIS <MODE CAST> DOES NOT MA
170H ITS A POSTERIORI MODE. )
RETURN
166 CALL OUTEVAL (P,REF,OUTCODE,7,1000-Z)
   I=I-1
   OPERAND(I)=ITEMP
RETURN

ENTRY PSI49A1

THE N(I) MODE ASSOCIATED WITH A <VOID CAST> IS 0, AND WILL CAUSE AN E
MESSAGE FROM ANY CALL OF SAMETYP THAT TAKES A ZERO AS ONE OF ITS PARA

P=P+1
N(I)=0
OPERAND(I)=0
RETURN

ENTRY PSI49A2

DEREFERENCEING AND DPROCEDURING MAY TAKE PLACE HEREF.

P=P+1
ITEMP=CONVERT(P,2R T)
IF (ITEMP.EQ.OPERAND(I)) GO TO 167
   CALL OUTCDX (Z,OPERAND(I),1M*,ITEMP,10,H, )
167 IF (SAMETYF(N(I),N(I-1),REF)) GO TO 168
   CALL ERROR (110H A PRIORI MODE OF THE <MODE CAST> DOES NOT MATCHITS
1 A POSTERIORI MODE. )
RETURN
168 CALL OUTEVAL (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 ITEMP=IFETCH(1)
   CALL SETIND (N(I),ITEMP,ITEMP1)
   CALL SETIND (3RPRC,ITEMP,3)
   N(I)=ITEMP


ENTRY PSI15B1
K=K+1
CODELOC(K)=Z
RETURN
ENTRY PSI16C
ITEMP=IFETCH(1)
CALL SETIND (N(I),ITEMP,2)
CALL SETIND (N(I-1),ITEMP,1)
CALL PUTIND (BYTE(CONT(N(I-1)),10),N(I),10)
I=I-1
N(I)=ITEMP
RETURN
ENTRY PSI77A1
IN THE PROCEDURE CALL, THE <PRIM RY> MUST BE LOWERED TO ITS FIRST MOD
A PROCEDURE WITH PARAMETERS.
NI=N(I)
ITEMP=LOCF(ITEMP4)
IF (NI,GT,0) GO TO 173
CALL ERROR (110MPROCEDUPE CALL ATTEMPTED ON A NON-PROCEDURE <PRIM
RY>.)
RETURN
ITEMP=CONT(NI)
ITEMP4=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
ITEMP5=IFETCH(1)
CALL SETING (ITEMP2,ITEMP5,3)
CALL SETIND (ITEMP5,ITEMP5,2)
ITEMP3=ITEMP5
NI=LINK(ITEMP2,2)
GO TO 173
IF (ITEMP1,NE,3RPAC) GO TO 172
IF (LINK(ITEMP,2,GT,0) GO TO 176
ITEMP2=3RPAC
GO TO 174
REF=ITEMP4
P=P+1
ITEMP4=CONVERT(P,2R T)
IF (OPERAND(I),EQ,ITEMP4) GO TO 177
CALL OUTGDA4 (2,OPERAND(I),1H*,ITEMP4,10H, //)
OPERAND(I)=ITEMP4
TYPE(I)=6XSTORED
N(I)=NI
CALL OUTEVAL (P,REF,OUTCODE,Z,1000-2)

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 OUTCOA4 (Z,0,0,ITEMP4,10H1 //)
TYPE(I)=7
CALL SETDIR (P+1,TYPE(I),2)
RETURN

ENTRY PSI7881

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

178 ITEM=R=LINK(TYPE(I-1),2)
ITEMP3=50
ITEMP2=ITEMP3+ITEMP2
IF (ITEMP3,0) OPERAND(I) GO TO 179
RETURN

179 P=ITEMP=1
CALL OUTCOA3 (Z,10HRETURN N RJ,10H1 3047), ,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.

IF (ITEMP3,0) RETURN
S=S+1
ITEMP2=ITEMP2+ITEMP2
CALL OUTCOA2 (Z,ITEMP2,10H1 //)
N(I)=ITEMP2
CALL SETDIR (ITEMP2,TYPE(I),2)
RETURN

ENTRY PSI788A1

ABBREVIATED VERSION OF PSI7881 WITHOUT INSERTION OF THE NEXT LABEL IN
OUTPUT CODE.

ITEMP3=0
GO TO 178

ENTRY PSI7882

LINKS TOGETHER THE LIST OF LABELS AND MODE LINKS.
CALL SETINO (OPERAND(I),OPERAND(I-1),1)
 16010
I=I-1
 16020
RETURN
 16030

ENTRY PSI77A2
 16040

OPERAND(I) CONTAINS THE LIST OF ENTRY LABELS AND DATA TYPES FOR EACH
 16040
ACTUAL PARAMETERS. P WAS RESET TO ITS RETURN LEVEL - 1 BY PSI78A1. LH
 16090
SAVED BEFORE PARAMETER EVALUATION, SO THAT ITS INITIAL VALUE BEFORE ST
 16100
PARAMETERS ON IT CAN BE RETRIEVED TO PASS TO THE PROCEOF, AT END OF
 16110
 R EVALUATION. CHECK FOR SAME LENGTH OF P-LISTS. TYPE(I-1) CONTAINS TH
 16120
 TO INSERT A JUMP OVER THE PARAMETERS, AND N(I-2) CONTAINS THE A POSTE
 16130
 PROEDURE MODE.
 16140

ITEM=TYPE(I-1),AND,.7777778
 16150
S=S+1
 16160
ITEM1=CONVERT(S,2R L)
 16160
OUTCODE[ITEM1]=ITEM1
 16170
OUTCODE[ITEM2]=10H+1
 16170
CALL OUTCDA2 (Z,ITEM1,10H+1           //
 16180
P=P+1
 16190
N=CONVERT(P,2R T)
 16200
CALL OUTCDA4 (Z,7HSAVE U,,LHCURR,5H+1#U,,10HSAVE U,, //
 16210
LOWS<ACTUAL PARAMETERS> TO A POSTERIORI LEVEL, THEN STORE IN TURN IN
 16220
ITEM5=0
 16230
COUNT OF PARAMETERS
 16240
ITEM1=OPERAND(I)
 16250
ITEM3=LINK(CONT(N(I-2)),1)
 16260
ITEM1 AND ITEM3 ARE, RESPECTIVELY, LINKS TO THE A PRIORI AND A POSTE
 16270
PARAMETER LISTS.
 16280
180 ITEM2=CONT(ITEM1)
 16300
ITEM2=CONT(ITEM3)
 16310
ITEM2=ITEM5
 16320
IF (SAME(TYPE(ITEM2,2),LINK(ITEM4,2),REF)) GO TO 181
 16330
CALL ERROR(110 THE PARAMETER IN THE JTH POSITION OF THE PROCED
 16340
URE CALL DOES NOT MATCH ITS A POSTERIORI MODE. J IS...
 16350
) CALL ERROR1(CONV(ITEM5,2R ))
 16360
GO TO 180
 16370
181 CALL OUTCDA4 (Z,10HSAVE RETUR,10H,J+2#RETU,3HRC,,CONT(ITEM1+1),1
 16400
10H+1#POP RETU,10HRC,           //
 16410
CALL OUTEVAL (P,REF,OUTCODE,Z,1000-Z)
 16440
CALL OUTCDA4 (Z,LHCURR,3H+1#LHCURR,10H,           //
 16490
CALL OUTCDA4 (Z,NI,1H+1#LHCURR,10H,           //
 16500
ITEM1=LINK(ITEM2,1)
 16510
ITEM3=LINK(ITEM4,1)
 16520
IF ((ITEM1,GT.0) AND,(ITEM3,GT.0)) GO TO 180
 16530
IF ITEM1,EQ.16300) GO TO 162
 16540
CALL ERROR (110 HILLEGAL PROCEDURE CALL--THE NUMBER OF ACTUAL AND F
NORMAL PARAMETERS DOES NOT MATCH. )
I=I-2
RETURN

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

182 CALL OUTCDA4 (Z, 8HPDP U, U*, 10HPARAMETERS, 10H, POP U, //)
I=I-2
CALL OUTCDA4 (Z, OPERAND(I), 8H(18-35)*, 9HDEFLEVEL, 6H, //)

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

CALL OUTCDA2 (Z, 10HENVIRONMEN, 10HTSAVE, //)
CALL OUTCDA6 (Z, 10HSAVE RETURN, 2HM, OPERAND(I), 10H((#17)*JUM, 10HP, J
12=RETUR, 10HRN, //)
CALL OUTCDA5 (Z, 8HJUMP), , 10HPOPRETURN, 10HRESTORE EN, 10HENVIRONMEN
17, 10H, //)
CALL OUTCDA3 (Z, 8HVALUE *, OPERAND(I), 10H, //)
TYPE(I)=6RSTORED
P=P-1
RETURN

ENTRY PSI73AI

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

P=P+1
ITEM=CONVERT(P, 2R T)
$=S+1
ITEM1=CONVERT(S, 2R L)
CALL OUTCDA5 (Z, 1H*, ITEM1, 1H*, ITEM1, 10H, //)
CALL OUTCDA6 (Z, 10HPROC LEVEL, 1H*, ITEM1, 10H(18-35), //)
CALL OUTCDA6 (Z, 10HPROC LEVEL, 1H*, ITEM1, 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)=ITEM
TYPE(I)=6RSTORED

N(I) SAVES THE LOCATION IN CODE FOR THE OVERJUMP
CALL PROCGRN(P, 0)

ENTERS THE HASH TABLE AND DOES BLOCK BEGIN BOOKKEEPING
RETURN
ENTRY PSI73A2

ITEM=IFETCH(1)
CALL SETIND(3PRG,ITEMP,3)
CALL SETIND(N(I),ITEMP,2)
CALL OUTCD2A4 (Z,OPERAND(I),10H VALUE, ,13H(RETURN),,10H
17180
17190
S=S+1
ITEMP1=CONVERT(S,2R L)
I=I-1
ITEMP2=N(I)
OUTCODE(ITEMP2,1)=ITEMP1
CALL OUTCD2A2 (Z,ITEMP1,10H)
17240
17250
ITEMP3=ITEM
17260
17270
CALL PRCDND(P,0)
RETURN
C
C ****************************
C END

SUBROUTINE OUTEVAL (P,REF,OUTCODE,Z,MAX)
INTEGER P,REF,OUTCODE,Z,CONT,CONVERT
DIMENSION OUTCODE(MAX,6)
ITEMP=CONVERT(P,2R T)
ITEMP3=LINK(CONT(REF),2)
C
C REF POINTS TO THE LIST CREATED BY SAMETYP CONTAINING INFORMATION
C ABOUT THE CORRECT SEQUENCE FOR DEPROCEDURING AND DEREFERENCING
C THE EXPRESSION CURRENTLY BEING EVALUATED.
C
C IF (ITEMP3.EQ.0) RETURN
C IF (ITEMP3.EQ.3) GO TO 102
C
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
C ITEMP4=CONVERT(LINK(CONT(REF),3),2R )
C CALL OUTCD2A4 (Z,ITEMP4,10H NUMBER OF,10H PARAMETERS,8H, //
C P=P-LINK(CONT(REF),3)
C ITEMP=CONVERT(P,2R T)
C CALL OUTCD2A4 (Z,10H SAVE RETUR,3HN, ,10H ITEMP#RJ( ,10H30#47), //
C
C CONTROL IS PASSED TO THE PROCEDURE9
C
C CALL OUTCD2A4 (Z,10HJ#RETURN,10H, RJ,, //)
C CALL OUTCD2A5 (Z,10HPOP,10H VALUE RETUR,3HN#,ITEMP,10H,
C 1 //)
C CALL OUTCD2A2 (Z,10HP#RETURN, ,10HVALUE, //)
C REF=LINK(CONT(REF),1)
C G0 TO 101
C
C THE NUMBER OF REFERENCE LEVELS IS FOUND IN LINK 3 OF THE WORD POINTED
C TO BY REF.
C
C 1Q2  ITEM4=LINK(CONT(REF),3) 350
   GO TO 104 360
103  ITEM4=ITEM4+1 370
104  CALL OUTCODE (z,ITEM4,ITEM2,ITEM4,ITEM4,ITEM10H, 380
      //)
      IF (ITEM4.GT.0) GO TO 107 390
      REF=LINK(CONT(REF),1) 400
      GO TO 101 410
      END 420
      INTEGER FUNCTION MINIMUM(NI) 430
      LOGICAL SAMEPYP 440
      INTEGER REF,CONT 450
      ITEM1=CONT(NI) 460
      LINK1=LINK(ITEM1,2) 470
      ITEM2=LINK(ITEM1,1) 480
101  ITEM1=CONT(ITEM1) 490
      LINK2=LINK(ITEM1,2) 500
      IF (SAMEPYP(LINK1,LINK1,REF)) GO TO 102 510
      IF (SAMEPYP(LINK2,LINK1,REF)) GO TO 103 520
      CALL ERROR (62) 530
      RETURN 540
102  LINK1=LINK2 550
103  ITEM2=LINK(ITEM1,1) 560
      IF (ITEMF.GT.0) GO TO 101 570
      MINIMUM=LINK1 580
      RETURN 590
      END 600
      SUBROUTINE BALANCE (LINK1,ITEM1,ITEM2,S,P0,Z,OUTCODE) 610
      DIMENSION OUTCODE(5000,6) 620
      LOGICAL SAMEPYP 630
      INTEGER S,P,Z,OUTCODE,CONT,P0 640
      ITEM3=LINK(CONT(ITEM2),2) 650
101  S=S+1 660
      ITEM=CONVERT(S,2R L) 670
      P=P+1 680
      CALL STRING (ITEM,ITEM1+2) 690
      CALL OUTCODE (z,ITEM,ITEM10H, 700
      CALL OUTCODE (z,ITEM,ITEM10H, 710
      CALL OUTCODE (z,ITEM,ITEM10H, 720
      CALL OUTCODE (z,ITEM,ITEM10H, 730
      CALL OUTCODE (z,ITEM,ITEM10H, 740
      CALL OUTCODE (z,ITEM,ITEM10H, 750
      CALL STRING (Z+1,ITEM1+3) 760
      HERE, THE ESCAPE ADDRESS IS PUT INTO THE THIRD LIST ELT., DATUM. 770
      SEE P$18473 FOR DETAILS. NOTE THAT THE ROWS OF CLAUSE USE TICODELOC(K 780
      AS THE VALUE RETURN ABOVE. 790
      CALL OUTCODE (z,0,ITEM1+2) 800
      ITEM1=CONT(ITEM1) 810
      ITEM2=LINK(CONT(ITEM2),1) 820
      IF (ITEM1.GT.0) GO TO 101 830
IF (ITEMP2.EQ.0) RETURN
   CALL ERROR110INCOMP, S BALANCE SUBROUTINE,
   STOP
RETURN
END
SUBROUTINE LOOP82D (LINK, LABEL, OUTCODE)
DIMENSION OUTCODE(5000,6)
INTEGER OUTCODE
   ITEMPL=INHALT(LINK+3)
101 OUTCODE(ITEMP,1)=LABEL
   ITEMPL=LINK(CONT(ITEMP,1))
   IF (ITEMP, ED.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 POSTERIO
   STRUCURE, ROW IS THE A PRIORI MODE STRUCTURE, OPRANDI IS THE LIST OF
   CLAUSE ENTRY LABELS.
   ITEMP3=CONVERT(P, 2R T)
   T=OPRANDI
   II=STANDARD
   IZ=ROW
   P=P0
102 J=1,LENGTH
   ITEMP1=CONT(I1)
   ITEMP2=CCONT(I2)
   LINK1=LINK(ITEMP1, 2)
   LINK2=LINK(ITEMP2, 2)
   IF (SAMEYPEP(LINK1, LINK2, REF)) GO TO 101
   CALL ERROR110NHIME FOLLOWING <ROW OF CLAUSE> FIELD DOES NOT
   MATCH ITS A POSTERIORI MODE.
   CALL ERROR1(CONVERT(J, 2R ))
   IF (SAMEYPEP(LINK2, LINK1, REF)) GO TO 101
   CALL OUTAL3 (Z, 10HSAVE RETUR, 10HN, J+2=RETU, 10HRN, )
   CALL OUTD2A (Z, CONT(I0+1), 10H,
   CALL OUTEVAL (P, REF, OUTCODE, Z, 1000-7)
   CALL OUTD3A (Z, 6HV+1=U, ,ITEMP3, 10H= U, )
   GO TO 101
   i0=CONT(I0)
   I1=LINK(ITEMP, 1)
102 I2=LINK(ITEMP, 2)
   CALL OUTED3A (Z, 6HV+1=U, ,ITEMP3, 10H= U, )
   CALL OUTD2A (Z, CONT(I0+1), 10H,
   CALL OUTEVAL (P, REF, OUTCODE, Z, 1000-7)
   CALL OUTD3A (Z, 6HV+1=U, ,ITEMP3, 10H= U, )
   CALL ERROR1(CONVERT(J, 2R ))
   CALL OUTED3A (Z, 6HV+1=U, ,ITEMP3, 10H= U, )
   CALL OUTD2A (Z, CONT(I0+1), 10H,
   CALL OUTEVAL (P, REF, OUTCODE, Z, 1000-7)
   CALL OUTD3A (Z, 6HV+1=U, ,ITEMP3, 10H= U, )
   RETURN
END
INTEGER FUNCTION IMASH(NAME, MODE)
   DIMENSION NAMES1(197,2), NAMES2(197,2), HASH1(197,2), HASH2(197,2)
1. STORAGE(50), PSTORE(50)
2. INTEGER HASH1, HASH2, STORAGE, BLOKNUM, PRLEVEL, PRCOUNT, BLCOUNT, PSTORE
3. CONVET
4. LOGICAL SAME TYPE
5. ENTRY INITIAL
6. CALLED ON ENTRY TO THE COMPIILATION OF A NEW PROGRAM.
7. PRLEVEL=0
8. BLOKNUM=0
9. PRCOUNT=0
10. MBCOUNT=0
11. HPCOUNT=0
12. L=IFETCH(1(0)
13. INITIALIZES ALL COMPILE TIME LISTS
14. RETURN
15. ENTRY FINAL
16. CALLED WHEN COMPILER REQUESTS TALLY OF NAMES IN NAMES AND PROC TABLE
17. NAME=MBCOUNT
18. MAXIMUM NAMES IN STATIC ENVIRONMENT
19. MODE=HPCOUNT
20. MAXIMUM NO. OF NAMES IN PROCEDURE ENVIRONMENT.
21. RETURN
22. ENTRY BLOKENN
23. BLOKNUM=BLOKNUM+1
24. STORAGE(0:BLOKNUM)=BLCOUNT
25. CALL SETDIR(0:PRCOUNB,STORAGE(BLOKNUM),2)
26. CALL SETDIR(NAME,STORAGE(BLOKNUM),3)
27. L=IFETCH2(0)
28. COMPILATE TIME LISTS BOOKKEEPING
29. RETURN
30. ENTRY BLOKEND
31. L=BLCOUNT-STOREAGE(BLOKNUM)+1
32. IF (L.LT.0) GO TO 103
33. DO 102 I=0,L
34. J=NAME(1:BLCOUNT-I,1)
35. L=HASH(HASH1,1)=0
36. BLCOUNT=STOREAGE(BLOKNUM).AND.777778
PRCOUNT=LINK(STORAGE(BLOKNUM),2)
   NAME=LINK(STORAGE(BLOKNUM),3)
104 L=IPETCH(0)
   CALLS THE LIST GARBAGE COLLECTOR.
   RETURN
   ENTRY PROCEGN
   PRLEVEL=PRLEVEL+1
   PSTORE(PRLEVEL)=PRCOUNT
   CALL SETDIR(NAME,PSTORE(PRLEVEL),2)
   GO TO 101
   ENTRY PROCEND

DELETES COMPILE TIME NAMES IN CURRENT PROCEDURE BLOCK, BUT LEAVES THE
TIME LOCATIONS INTACT.

L=PRCOUNT-1-LINK(PSTORE(PRLEVEL),1)
   NAME=LINK(PSTORE(PRLEVEL),2)
105 IF (L.LT.0) GO TO 106
   DO 105 I=0,L
   J=NAMES2(PRLEVEL,1)
   106 HASH2(J,1)=0
   GO TO 105
   ENTRY IHASHA

LOOK UP NAME AND MODE, OR LABEL, AND LIST OF LOCATIONS TO BE FILLED, AND
RUN-TIME ALIAS PRECEDED BY ~.

LASTUSE=0
   NAME
   IF (PRLEVEL.EQ.0) GO TO 111
   DO 110 L=0,196,1
      I=MOD(I+47*L,197)+1
      IA=HASH2(I,1)
      IF (IA.NE.0) GO TO 108
      IF (LASTUSE.GT.0) GO TO 107
      GO TO 111
      107 IHASHA=CONVERT(LASTUSE,2*P)
      MODE=LOCF(NAMES2(LASTUSE,2))
      RETURN
   108 IF (IA.NE.NAME) GO TO 109
      LASTUSE=HASH2(I,2)
   109 CONTINUE
   110 CONTINUE
   IF (LASTUSE.GT.0) GO TO 107

NAME IS NOT WITHIN PROCEDURE ENVIRONMENT.

111 00 115 L=0,196,1
I=MOD(I+47*L,197)+1
IA=HASH1(I,1)
IF (IA.EQ.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 ERROR(111)
112 IHASHA=CONVERT(LASTUSE,2R^V)
MODE=LOCF(NAMES1(LASTUSE,2))
RETURN
113 IF (IA.NE.NAME) GO TO 114
LASTUSE=HASH1(I,2)
114 CONTINUE
115 CONTINUE
IF (LASTUSE.GT.0) GO TO 112
ENTRY IHASHB
ENTER NAME AND MODE ON NAMES TABLE, AND RETURN RUN-TIME ALIAS PREFIXED
I=NAME
DO 116 L=0,196,1
I=MOD(I+47*L,197)+1
IA=HASH2(I,1)
IF (IA.EQ.0) GO TO 117

ENTER THE NAME.
116 CONTINUE
CALL ERROR (110) COMPIL-NAME TABLE OVERFLOW DURING A PROCEDURE
1E DECLARATION.
RETURN
117 PRECOUNT=PRECOUNT+1
IF (PRECOUNT.GT.HPCOUNT) HPCOUNT=PRECOUNT
HASH2(I,1)=NAME
HASH2(I,2)=PRECOUNT
NAMES2(PRECOUNT,1)=I
NAMES2(PRECOUNT,2)=MODE
MODE=LOCF (NAMES (PRECOUNT,2))
FOR USE BY LABELS.

IHASHB=CONVERT(PRECOUNT,2R^P)
RETURN
118 DO 119 L=0,196,1
I=MOD(I+47*L,197)+1
IA=HASH1(I,1)
IF (IA.EQ.0) GO TO 120
119 CONTINUE
CALL ERROR (110) COMPIL-NAME TABLE OVERFLOW OUTSIDE OF A PROCEDURE DECLARATION.
RETURN
120 BLCOUNT=BLCOUNT+1
IF (BLCOUNT.GT.BLDEPTH) BLDEPTH=BLCOUNT
72

\[ \text{ENTRY INASH2} \]

\[ \text{THIS FUNCTION LOOKS UP OPERATORS IN THE NAME TABLE. WE ASSUME THAT AN} \]
\[ \text{OPERATOR NOT PRESENT ON THE TABLE IS EITHER INVALID OR WIRED IN TO THE} \]
\[ \text{RUN-TIME PILOT SYSTEM.} \]

\[ \text{LASTUSE} = 0 \]
\[ \text{I} = \text{NAME} \]
\[ \text{IF} (\text{LEVEL} > 0) \text{ \text{GO TO 124} \}
\]
\[ 127 \quad L = 0, 196, 1 \]
\[ 128 \quad I = \text{MOD}(I + 47 * L, 197) + 1 \]
\[ 129 \quad IA = \text{HASH2}(I, 1) \]
\[ 130 \quad \text{IF} (IA \neq 0) \text{ \text{GO TO 121} \}
\]
\[ 131 \quad \text{IF} (\text{LASTUSE}, GT, 0) \text{ \text{GO TO 127} \]
\[ 132 \quad \text{GO TO 124} \]
\[ 133 \quad TA = \text{HASH2}(I, 2) \]

\[ \text{COMPARE OPERAND MOTES.} \]
\[ \text{IF} (\text{LINK}(\text{CONT}(\text{NAME}(I, 2), 1)) \]
\[ \text{IF} (\text{IA} = \text{NAME}) \text{AND} ((\text{NOT}, \text{SAME}, \text{TF}(\text{MOTION}, I, 2))) \text{ \text{GO TO 122} \]
\[ \text{LASTUSE} = \text{HASH2}(I, 2) \]
\[ \text{CONTINUE} \]
\[ \text{IF} (\text{LASTUSE}, GT, 0) \text{ \text{GO TO 107} \]
\[ \text{CONTINUE} \]
\[ \text{IF} (\text{LEVEL} > 0) \text{ \text{GO TO 124} \}
\]
\[ 136 \quad L = 0, 196, 1 \]
\[ 137 \quad I = \text{MOD}(I + 47 * L, 197) + 1 \]
\[ 138 \quad IA = \text{HASH1}(I, 1) \]
\[ 139 \quad \text{IF} (IA \neq 0) \text{ \text{GO TO 125} \]
\[ 140 \quad \text{IF} (\text{LASTUSE}, GT, 0) \text{ \text{GO TO 112} \]
\[ 141 \quad \text{LASTUSE} = 0 \]
\[ \text{RETURN} \]
\[ \text{IF} (\text{LINK}(\text{CONT}(\text{SAME}, (I, 2), 1)) \]
\[ \text{IF} (\text{IA} = \text{NAME}) \text{AND} ((\text{NOT}, \text{SAME}, \text{TF}(\text{MOTION}, I, 2))) \text{ \text{GO TO 126} \]
\[ \text{LASTUSE} = \text{HASH1}(I, 2) \]
\[ \text{CONTINUE} \]
\[ \text{IF} (\text{LASTUSE}, GT, 0) \text{ \text{GO TO 112} \]
\[ \text{RTUP} \]
\text{END}