stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 93/77 NOVEMBER

E. DE JONG

A PROPOSAL FOR THE TRANSLATION OF JANUS INTO STANDARD FORTRAN

Preprint

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

AMS(MOS) subject classification scheme (1970): 68A15

ACM-Computing Reviews-categories: 4.12, 4.22

A proposal for the translation of Janus into Standard Fortran *)

by

E. de Jong

ABSTRACT

To investigate the possibility of the mechanical translation of programs written in the intermediate code Janus into Standard Fortran, some aspects of Janus are examined. Suggestions for solving the problems involved are formulated.

KEY WORDS & PHRASES: Janus, Standard Fortran, machine-independence, intermediate code, portability

*) This report will be submitted for publication elsewhere.

INTRODUCTION

Janus [1] is one of the languages that can be chosen as intermediate code for the translation of some high level language into a machine-independent language. The problems which may arise during the compilation of Janus give an impression of its usefulness in comparison with other such intermediate languages. Although Janus was intended to be translated into assembly language, it may be made more portable by distributing a translator from Janus to some generally available machine-independent language to enable a quick and dirty first bootstrap. Standard Fortran [2] is chosen as machine-independent language, since on almost every computer in the world there exists at least one implementation of Fortran. A compiler which produces Standard Fortran as final code can certainly be considered a portable one. It is reasonable to make a distinction between the global and the detailed proposals which are made in this paper. The global ones primarily concern the organization of the memory, the construction of activation records, etc.; the detailed ones define the translation of such things as conditional jump instructions and arithmetic operations. Although I originally tried to translate the Janus program by a one pass compiler, it gradually became clear that it is necessary to do the compilation in two passes. For example, it is impossible to translate a jump out of a Janus procedure by a one pass compiler. The translation of other Janus concepts such as global declarations and constant definitions will be easier. Fortran code is generated during the first pass while such things as the addresses within the operand stack and the number of entry-points are filled in during the second pass.

DEVIATIONS FROM STANDARD FORTRAN

Occasionally, for the translation of some Janus instruction it is necessary to deviate from Standard Fortran. For example, some Janus instructions require values of unknown type to be copied. In such cases I interpret this variable as being a real one; although it is not Standard Fortran, on most implementations it is possible to copy integer and logical values as if they were reals. Such use of real numbers has been limited to specific run-time routines which can easily be replaced if necessary for a specific installation.

ASSUMPTIONS MADE ABOUT JANUS

Although the intermediate code Janus is defined in [1], I must emphasize that it is a preliminary definition. In many places in this document it is not clear what is meant and thus I had to guess to the intentions of the authors. The usefulness of some concepts in the Janus definition is at least questionable. For example, I am not completely certain of the interpretation of the ARVND-line; I have interpreted this line like the ELMV-line. The occurrence of symbol in this line seems useless to me. I have been forced to make assumptions as to the meaning of

Janus in many dark corners.

ORGANIZATION OF MEMORY

The memory available for the translation of the Janus program is divided in three parts: the space needed for constants, which is known before execution of the program, the activation record stack and the heap (dynamic storage). It is clear that these three categories must not overlap. The Janus memory is implemented as a Fortran one-dimensional array. Equivalence statements enable each element of this array to be occupied by an integer, a real or a logical value; it is therefore required that integer, real and logical values be represented by equal numbers of storage units. For this reason it is impossible to implement double precision constants in the same array. This implementation method implies that the fields of a Janus record and the elements of a Janus array occupy consecutive storage units in this Fortran array. The array is split as follows: the constants first, the stack and operand stack next, and the heap last. Because the total amount of memory needed by constants is known when execution of the program begins, the starting address of the stack is pointed to by an integer variable. When there are local declarations or procedure calls, this variable is increased or changed. Recursive procedure calling is possible in Janus, but not in Fortran. Janus procedures cannot be implemented as Fortran subroutines in a straightforward way. It does not seem sensible to solve this problem by implementing Janus procedures as so-called "open code" (the Janus program - including all Janus procedures - is translated into a Fortran program without subroutines). It is quite imaginable that there are Fortran compilers that cannot compile very large Fortran programs. It is better to translate each Janus procedure into a Fortran subroutine. This requires some special information within the activation record of each procedure to make recursive calling possible.

THE ACTIVATION RECORD

Each Janus procedure (and so each Fortran subroutine) requires an activation record. Within this record the available space is divided into four parts: 1. parameters (if any), 2. administration, 3. local variables and 4. space for operands and unnamed temporaries. If there are parameters, they are placed in the activation record first; I prefer this method to doing the administration first and the parameters later; for recursive calls this saves space, since no administration is needed before the procedure is actually called. The administration requires four integer storage units; I will discuss this part of the activation record in the following section. In the third component of the activation record space is reserved for the local variables and named temporaries as explicitly declared in the Janus program. Finally, the activation record contains the storage used for operands, unnamed temporaries, intermediate results, etc. One must carefully prevent overlap between the operand stack and the rest of the activation record. This may happen when intermediate results have to be stored, for example in newly declared (named) temporaries. If the operand stack and the stack touch each other in such a situation, overlap may occur. This can be avoided by reserving all explicitly declared storage of an activation record before determining where the operand stack begins. During the first pass the total number of storage units needed for explicitly declared variables is calculated; during the second pass the starting address of the operand stack is determined.

CALLING AND RETURNING FROM A PROCEDURE

The translation of a Janus program implies the creation of a "supervising" Fortran (main) program. It is not possible to call or to return from any subroutine without the intervention of the supervising program. Only one Fortran subroutine is active at the same time. A call causes the creation of the activation record of the subroutine to be called. As already mentioned, this activation record contains an administration part. This part consists of 4 integer storage units; the first integer indicates the starting address of the addressing environment in which the procedure to be called is declared (the static link); the second integer is the "procedure number". The supervising program uses this value to select the procedure to be called; therefore, all Fortran subroutines that are translations of Janus procedures must be numbered. The third integer indicates the starting address of the addressing environment of the calling procedure (the dynamic link). The last integer value determines the entry-point of the subroutine; when a subroutine is called, this value corresponds to the first executable statement; this integer value equals 1.
When a call is made the administration part of the subroutine to be called is filled in by the calling subroutine. Before control is given to the supervising program by the calling subroutine, the entry-point number in the administration part of the calling subroutine must be changed. This number must agree with the statement immediately following the call, so that upon return this statement is the first one to be executed. This statement must be labeled to enable it to be jumped to. I will discuss this in more detail at the translation of the RCEND-instruction. After all these values are determined, the supervising program calls the subroutine indicated by the second value within the new administration part; the fourth value (equal to 1) forces the first statement of the subroutine to be executed.

Returning from a subroutine means returning to the supervising program using the dynamic link to find the activation record of the caller. The supervisor gives control to the calling subroutine (determined by the second value in the administration part) and the first statement to be executed now is determined by the fourth value of the administration part. The code generated for this is discussed at the translation of the RETURN-instruction. It is worthwhile to note that a procedure which includes N calls to other (possibly different) procedures has at least N+1 entry-points. If a procedure includes a label which is jumped to from outside that procedure, such a label corresponds to an entry-point also.
In the foregoing I have not mentioned the elaboration of parameters, the further internal structure of the activation record stack, etc. These

subjects are discussed in a later section.

RECURSIVE AND NONRECURSIVE PROCEDURES

In my implementation no distinction is made between Janus recursive and nonrecursive procedures. Both are called as if they were recursive. It would have been possible to implement nonrecursive Janus procedures directly as Fortran procedures, that is to say a call to a nonrecursive procedure causes no intervention of the supervising program, but is called directly by the calling subroutine. However, this method is possible only when no nonrecursive Janus procedure includes a call to a recursive procedure. Such a call would have required a return to the supervising program. This problem could be solved by implementing a nonrecursive procedure as a "supervising subprogram". This solution would not have been more efficient than to consider each procedure to be recursive, and severe problems would arise in implementing external labels.

CONVENTIONS

Until now I have only described the global outline and architecture of the Fortran program as a result of the compilation of the equivalent Janus program. The next part of this paper describes how the various Janus instructions are translated. The following Fortran execution-time variables are used:

MEM	: a Fortran one-dimensional array of type integer, which is equivalent to the Janus memory.
BM,RM	: one-dimensional Fortran logical and real arrays, respectively, which are "equivalenced" with MEM by the statement: EQUIVALENCE(MEM(1), BM(1), RM(1)).
IND	: an integer variable, which stands for the Janus index register.
IBS	: an integer variable that implements the Janus base register.
MP	: an integer variable, which marks the beginning of the activation record of the current active subroutine. This variable is initialized when all constant definitions in the Janus program are processed.
I,J,K	: integer variables, which have no special meaning; they are used only in complicated calculations such as the transport of large parts of memory.
X	: a real variable used to store intermediate results.
N	: an integer variable used in the translation of Janus instructions that set the condition-code; it occurs as left operand of a relational expression.

Moreover, the following compile-time variables are used:

T1	: an integer variable counting the number of Janus procedures including the main program (initial value = 1) .
LA	: a one-dimensional array of type integer; for a Janus procedure with number N the array element LA(N) counts the number of entry-points of the equivalent Fortran subroutine.
EP	: a two-dimensional array of type integer; in a Janus procedure with number N, EP(N,K) contains the value of the Fortran label

number corresponding to the K-th entry-point of this procedure.

IT	: a one-dimensional integer array; the element IT(N) counts the number of storage units occupied by administration and variables that are explicitly declared in the Janus procedure with number N; during the second pass this variable is used to determine the starting address of the operand stack in the equivalent Fortran subroutine.
MDS	: an integer variable indicating the maximum number of storage units needed for declared variables in a Janus procedure.
PD	: an integer variable used to determine the addresses of explicitly declared variables.
DL	: an integer variable indicating the display level of the current procedure.
LC	: an integer label counter; for each Fortran subroutine the value of this variable is used when a Fortran label is generated.
ID	: an integer variable indicating the integral displacement from the beginning of the activation record.
TP	: an integer array; the element TP(N) indicates the number of storage units occupied by the parameters of the procedure with number N.
PT	: the array-element PT(N) contains <u>symbol</u> of the Janus procedure with number N.
LM,MS	: one-dimensional integer arrays used at the translation of record-mode definitions.
MX,CC	: integer variables used at the translation of record-mode definitions.
A1,A2,A3	: one-dimensional integer arrays used at the translation of composed values (APPENDIX 1).
N,K,T	: integer variables used at the translation of composed values.
BL	: one-dimensional integer array used as compile-time stack at the translation of BLOCK- and BLEND-instructions.
GC	: an integer variable used at the translation of COMMON DSECT lines.
SIZE	: an integer variable indicating the length of the Fortran memory available for the translated Janus program; the value of this variable is determined by the programmer of the Janus program.
C	: an integer variable counting the number of storage units occupied by constants (initial value = 1).

Expressions enclosed by " " are elaborated at compile-time, and the resulting value is put out. The variables occurring in such expressions are compile-time variables and do not appear in the code that is generated.

Words included within [and] are optional.

Words which are underlined represent more general Janus concepts, e.g. the notation <u>mode</u> is used when I want to discuss objects, whose mode is arbitrarily chosen. It is used primarily to indicate nonterminal symbols which occur in the Janus grammar.

Words which are underlined and enclosed by ′ ′ stand for the corresponding values that are stored in the symbol or identifier table.

Most Janus instructions cause Fortran code to be generated; in many cases the values of compile-time variables must be changed also; these compile-time changes are described in terms of Fortran statements preceded by C1 for changes during the first pass and by C2 for changes during the

second pass of the compilation.
In the description of the translation of the Janus instructions a compile-time identifier and symbol table are used. It is not meant that these tables are separated strictly; the only difference between these tables is that the data in the identifier table are accessible during the compilation of the whole Janus program; all data in the symbol table are lost each time a Janus FINISH-statement is encountered; these data can only be reached at the compilation of a Janus module.

OPERANDS

Most Janus instructions are composed of an operator and an operand. In the next sections each operator will be discussed separately; however, the handling of operands is the same for many operators. Therefore, attention is first paid to such things as _targets_, _cells_, _data-locations_ etc. All these objects can be classified in five categories.

1. _code-location_
2. _temporary_
3. _data-constant_
4. CONST _reference_
5. _variable_

I will now define a translation for each of these classes.

Code-location.

In a Janus program, code-locations may occur in three cases:

a. jumps
b. calls
c. case selections

Three kinds of code-locations are possible:

1. R _symbol_
2. X _xsymb_
3. Y _xsymb_

Firstly, I will discuss code-locations of the form: R _symbol_.
ad a. Since code-locations of this kind are within the current procedure, _symbol_ and an integer value indicating the Fortran statement number are stored in the symbol table. Forward reference of labels will be discussed at the translation of the jump instruction.
ad b. When a procedure is called or referenced (when a PROC-mode object is created), the number of this subroutine must be known. The symbol which occurs in the procedure heading is identical to the label of the code-location. This symbol and the number of the procedure have been stored in the symbol table.
ad c. Since a case-selection is translated into a row of conditional jumps, the same method of storing is used as in case a.
X _xsymb_ is interpreted as an entry-point of another Janus module. It is not possible to jump to another module, since the addressing environment is not known. Therefore code-locations of this kind are not

implemented. I feel the Janus definition must be revised on this point.
The occurrence of Y xsymb implies some "system" action to be performed. The implementation of such a code-location is left to the individual implementer, since it is not possible to decide for a particular machine what to do when a system symbol is encountered.

Temporary.

Since the lifetime of a temporary cannot exceed a basic block, the address of a named temporary is always within the current activation record. This means that only a displacement need be stored in the symbol table. For example, referring to an integer temporary is done by: MEM(MP+"DI"), where DI is the displacement as stored in the symbol table and MP marks the beginning of the activation record currently active.

Data-constant.

Three kinds of data-constants occur in the Janus program:

1. E expression
2. denotation
3. M identifier

ad 1. It is possible to include compile-time integer expressions in the Janus program; they are preceded by the letter E; in these expressions only (signed) integer values and int-variables occur. Although nothing is said in the Janus document about these int-variables, I must assume that their values are known in one way or another at compile-time (e.g. stored in the identifier table). It is most reasonable to generate a copy of the Janus expression in Fortran, in which the int-variables are replaced by the values represented by them.
ad 2. In Janus there are four kinds of denotations:

1. ACHAR character

2. CINT character

Since there are no constants of type character in Fortran, these two kinds of denotations are transformed into integer values. I have not chosen for an implementation by means of Hollerith constants, since in this way the comparison of two characters could fail because of the possibility of overflow when two Hollerith constants are compared. It is preferred to transform characters into integer values and back by two functions similar to the Pascal ′ord′- and ′chr′-functions. No distinction is made between ACHAR- and CINT-denotations.

3. AINT [-] digit-sequence

It will be clear that the optional sign and the digit-sequence are copied into an equivalent Fortran integer constant.

4. AREAL [-] digit-sequence E [-] digit-sequence

The equivalent Fortran constant can easily be determined; if DS1 denotes the first digit-sequence and DS2 the second then this constant is: [-]."DS1" E [-] "DS2". For example, AREAL 1E-2 is transformed into .1E-2.

ad 3. The value of a manifest is easily found, since the data-constant is stored in the compile-time manifest table when the manifest definition is encountered. By means of an index in this table the value involved is generated.

CONST reference.

All constants declared in the Janus program are stored in the constant part of the Fortran memory. The address of the constant is computed by adding the values of base and field- selector (if present) and the value of index, which is the value of expression multiplied by the value of mode that is stored in the symbol table; If a * is present, this value is increased by the value of IND multiplied by the value of mode; If a + is present this value is increased by the value of IND only.

Variable.

There are several kinds of variables in Janus; one kind is of the form: STATIC|COM reference.
The space declarations within a global declaration and the static declarations are interpreted as storage allocation in the constant part of the (Fortran) memory. Therefore, reference refers to an address in the constant part. All values contained within reference are interpreted in the same way as CONST reference.
Another possibility of a variable is: DISP′i′|LOCAL|PARAM reference. To reach variables declared in outer blocks, the prefix DISP′i′ is used. To compute the address of the variable involved, the display level of the current activation record (DL) need be known. The following code is generated:

```
if i = DL              : I = MP

if i = DL-1            : I = MEM(MP)

if i ≤ DL-2            : I = MEM(MP)
                         L = "DL"-′i′-1
                      C1 LC = LC+1
                         DO "LC" M = 1,L
                    "LC" I = MEM(I)
```

The integer variable I contains the value of the beginning of the activation record in which the variable is declared. By means of data stored in the symbol table, the displacement of the variable as denoted by reference is computed. The address of the variable is: I+"DI", where DI is the integral displacement within the activation record.

LOCAL reference.

A special case of DISP′i′ variables are those that are local to the current activation record; this means that their addresses are within the address space of this activation record. If DI denotes the displacement corresponding to reference computed from the beginning of the activation record, the address of the variable is: MP+"DI". Note that DI is negative if the variable is a parameter of a recursive procedure.

PARAM reference.

The address of the parameter is calculated. As mentioned before, in our implementation the parameters are placed before the beginning of the activation record. So all parameters are reached by means of negative displacements. The value of this displacement is known when the PAREND-instruction is encountered. Clearly, this displacement has to be computed from the start of the current activation-record; so the address of a parameter is: MP-"TP(T1)-DI". The base PARAM is only used for parameters of NONREC procedures.

BASED [reference].

The integer variable IBS corresponds to the Janus base register. If reference is absent, then the value of IBS is the address needed. If, however, reference is present, then the values of index and field-selector are added to the value of IBS. The address is denoted by IBS+"DI", where DI corresponds to the value of reference.

VALUES

Apart from the data-constants already discussed, it is possible to specify initial values at constant definitions or at static or common declarations. This initialization is implemented by DATA statements. There are two categories of composed values: record-values and array-values. A detailed description of the translation of FLDV-, ELMV-, RCVND- and ARVND-lines is given in APPENDIX 1.

FLDV mode STATIC|CONST|CSECT symbol [(expression)|[expression]] [value].

Since only array element names (and not array names) are allowed within a Fortran DATA statement, for each elementary value a DATA statement is generated. If no value is given, no DATA statement is generated. C is increased by the product of the value of mode and the value of expression, if any, and otherwise by 1. If value is not a data-constant and thus begins with a +, subsequent lines specify the value(s) of this field. If an expression is given, the field is an array of mode elements. The value is assigned to a number of elements determined by the value of expression. If value is a data-constant, then a Fortran DATA statement is generated; C is increased by the value of expression, if any, and otherwise by 1.
For example:

```
FLDV INT CONST A1(5) AINT 2.
```

is translated into:

```
        DATA MEM(7), MEM(8), MEM(9), MEM(10), MEM(11)/5*2/  ,
```

assuming that the value of C is 7. The fields of a Janus record are implemented as successive elements in the Fortran array.

RCVND mode CONST|STATIC|CSECT [symbol].

This Janus statement closes a record-value.

ELMV mode STATIC|CONST|CSECT [expression]|(expression) [value].

This Janus instruction is handled in nearly the same way as the FLDV-instruction. If value is a data-constant, then a DATA statement is generated. The length of the list of array element names depends on the value of expression. I refer to the example given at the FLDV-instruction. If value is a composed value (indicated by a +) subsequent lines specify the values for a number of elements determined by the value of expression. If no value is given, C is increased by the product of the value of mode and the value of expression. The elements of a Janus array are implemented as successive elements in the Fortran array. No distinction is made between packed and aligned Janus arrays.

ARVND mode CONST|STATIC|CSECT [symbol] [expression]|(expression) [value].

This line indicates the end of an array-value; otherwise this statement is handled in the same way as the ELMV-instruction.

TRANSLATION OF INSTRUCTIONS

A Janus program is composed of Janus modules. However, the difference between a program and a module is not clear to me. No definition of program or module is given in the Janus document. It could be that a program and a module are the same things; in the description of several other Janus concepts these two words are freely interchanged.

Module.

START identifier.

This Janus instruction indicates the beginning of a Janus module. When a constant definition is encountered, the value(s) within that definition is (are) translated into (a) DATA statement(s) that are put out on a separate constant file; this constant file is built up during the first pass of compilation; it is compiled separately from the program file; at execution-time these two files are joined. All DATA statements are contained in a BLOCK DATA subprogram; when the first START-instruction is encountered the following code is generated:

```
          BLOCK DATA                                    1
          LOGICAL BM("SIZE")                            2
```

```
        REAL RM("SIZE")                                  3
        COMMON/H/ MP, MEM("SIZE")                        4
        EQUIVALENCE(MEM(1), BM(1), RM(1))                5
```

FINISH.

This instruction marks the end of a Janus module. The symbol table is cleared; only the data stored in the identifier table are saved. If it is the last instruction of the program, the number of Janus procedures is known; then it is possible to generate code for the supervising program. The following code is then generated:

```
        C  PROGRAM SPV(OUTPUT,TAPE6 = OUTPUT)            1
           LOGICAL BM("SIZE")                            2
           REAL RM("SIZE")                               3
           COMMON/H/ MP,MEM("SIZE")                      4
           EQUIVALENCE(MEM(1),BM(1),RM(1))               5
           MEM(MP-3)=1                                   6
           MEM(MP) = MP-4                                7
           MEM(MP+1) = 2                                 8
           MEM(MP+2) = MP-4                              9
           MEM(MP+3) = 1                                10
         1 I = MEM(MP+1)                                11
           GO TO (11,12,13,...,"10+T1"),I               12
        12 CALL P2                                      13
           GO TO 1                                      14
        13 CALL P3                                      15
           GO TO 1                                      16
           ...
           ...
           ...
   "10+T1" CALL P"T1"                                   17
           GO TO 1                                      18
        11 STOP                                         19
           END                                          20
```

The beginning letter C on line 1 may be omitted on some implementations, such as the Cyber 73 of Control Data Corporation. Lines 2-5 declare the Fortran array which corresponds to the Janus memory. Line 6 tells us indirectly what to do when execution is finished. The integer 1 will eventually cause line 12 to jump to line 19. Lines 7-10 specify the contents of the administration of the activation record of the main program. Line 7 fills in the static link, line 8 tells us, that it is the main program which must be called, line 9 fills in the dynamic link and line 10 specifies the entry-point within the main program. When the last FINISH-instruction is encountered storage has been reserved for all constant definitions. A DATA statement to initialize MP is generated when all constant definitions, static and global declarations have been processed. Dynamic allocated storage is placed at the end of the Fortran array; two pointers are initialized. On the constant file the following code is generated:

```
        DATA MP/"C+4"/                                   6
        DATA MEM("SIZE")/"SIZE-1"/,MEM("SIZE-1")/0/       7
        END                                               8
```

Note that 4 storage units remain unused; they are considered to be needed for the activation record of the supervising program.

Declarations.

GLOBAL xsymb ENTRY symbol.

There is no visible Fortran translation for this instruction. Only some administration is done. As symbol is the name of a Janus procedure, this symbol refers to an entry in the compile-time symbol table. This entry will now be referred to by the identifier xsymb. In other words, the entry pointed to by symbol is associated with xsymb; since xsymb is stored in the identifier table, this entry is known outside the module.

GLOBAL [xsymb] CSECT symbol.

This instruction begins a COMMON block. This means that the space declarations which follow are treated as if they were constant definitions. Storage for these global variables is allocated within the constant part of memory. The current value of C is associated with xsymb and symbol; this value and xsymb are stored in the identifier table; symbol and the value of C are stored in the symbol table.

GLOBAL [xsymb] DSECT symbol.

If xsymb is in the identifier table, the value of xsymb is assigned to GC and to symbol, which is stored in the symbol table; GC is used to determine the addresses of the symbols occurring in the following SPACE lines.
If xsymb is not in the identifier table, all Janus lines up to and including the corresponding ENDBLK-instruction are ignored. The translation of these lines is performed during the second pass.

SPACE mode CSECT declarator[value].

This global variable declaration effectively causes the reservation of space. The amount depends on the mode which is specified; symbol (of declarator) and the value of C are stored in the symbol table; if no value is given, C is increased by the product of the value of mode and the value of expression (of declarator), if any. If value is present and begins with a +, the new value of C is determined during the translation of FLDV- or ELMV-instructions. The following variables are initialized:

```
        C1 T=1
        C1 K=0
        C1 N=0
```

If value is a data-constant (and thus mode is a primitive-mode), a DATA statement is generated (on a separate file for constants); the number of array element names depends on the value of expression. C is increased by the value of expression, if any, and otherwise by 1.

SPACE mode DSECT declarator.

If this statement is not ignored, symbol (of declarator) and the value of GC are stored in the symbol table; GC is increased by the product of the value of mode and the value of expression, if any.

ENDBLK CSECT|DSECT symbol.

This instruction marks the end of a Janus COMMON block. No code is generated and no administration takes place.

BEGIN [primitive-mode] DISP'i'|MAIN|NONREC symbol.

This instruction is the first of a Janus procedure declaration. If primitive-mode is present, the procedure delivers a value of that type, that is to say an integer, real, boolean, character, procedure or address value. Nonrecursive procedures do not have local declarations; only temporary and static declarations are possible. The addressing environment of a recursive procedure is determined by the integer 'i' as occurring in DISP'i'. 'i' indicates the number of display pointers needed for accessing nonlocal variables. The display level must be recorded, since it is used in the computation of the addresses of the variables in the procedure. The value of T1 and symbol are stored in the symbol table. The following code is generated:

```
C2 T1 = T1+1
C1 T1 = T1+1
C1 MDS = 4
C1 PD = 4
C1 DL = 'i'|0|DL+1
C1 LC = 1
C1 LA(T1) = 1
C1 PT(T1) = "symbol"
C1 EP(T1,1) = 1
C1 TP(T1) = 0
   SUBROUTINE P"T1"
```

PARAM mode DISP'i'|MAIN|NONREC symbol [(expression)|[expression]].

No Fortran code need be generated by this Janus instruction. symbol and the value of TP(T1) are stored in the symbol table; TP(T1) is increased by the product of the value of mode and the value of expression, if any, and 1 otherwise.

PAREND [primitive-mode] DISP'i'|MAIN|NONREC symbol.

This instruction marks the end of the parameter list; so at this moment the total number of parameters and the number of storage cells needed for them are known. This implies that the addresses of all parameters can now be computed; each parameter now corresponds with a unique address recorded in the symbol table. As mentioned before, the parameters are accessible by a negative displacement with respect to the beginning of the activation record. For that reason the address of each parameter cannot be computed before the total number of storage units needed is known; the current value of TP(T1) is used at parameter references. The following code is generated:

```
          LOGICAL BM("SIZE")
          REAL RM("SIZE")
          COMMON/H/MP, MEM("SIZE")
          EQUIVALENCE(MEM(1), BM(1), RM(1))
          I = MEM(MP+3)
          GO TO("EP(T1,1)",...,"EP(T1,LA(T1))"),I
        1 CONTINUE
       C2 ID = IT(T1)-1
```

The total number of statement numbers depends on the number of calls and the number of labels jumped to from outside this procedure; EP(T1,K) is the value of the Fortran statement label corresponding to the K-th entry-point of the current procedure.

END [<u>primitive-mode</u>] DISP'i'|NONREC|MAIN <u>symbol</u>.

The following code is generated:

```
          END
       C1 IT(T1) = MAXO(MDS,PD)
```

SPACE <u>mode</u> DISP'i' <u>symbol</u> [(<u>expression</u>)|[<u>expression</u>]].

The effect of this "local declaration" is to allocate storage within the activation record of the current procedure. The number of storage units depends on <u>mode</u> and <u>expression</u>. <u>symbol</u> and the value of PD are recorded in the symbol table. PD is increased by the product of the value of <u>mode</u> and the value of <u>expression</u>. No code is generated.

BLOCK 'i' DISP'j'.

This Janus instruction indicates a local block within the activation record of the current DISP'j' procedure. Therefore, a compile-time pointer is set, so that by encountering the corresponding BLEND-statement, the storage reserved for local variables declared within this local block can be freed again. It is necessary to have a compile-time stack for this purpose. The following Fortran code is generated:

```
       C1 BL('i') = PD
```

BLEND 'i' DISP'j'.

As already mentioned, the pointer which has been set at the corresponding BLOCK-statement indicates which storage units are freed now.

```
       C1 MDS = MAXO(MDS,PD)
```

```
    C1 PD = BL('i')
```

TEMP primitive-mode DISP'i'|NONREC|MAIN symbol.

If this instruction is contained within a NONREC-procedure then this declaration is treated as if it were:
SPACE primitive-mode STATIC symbol.
Otherwise it is handled as:
SPACE primitive-mode DISP'i'|MAIN symbol.

RELEASE primitive-mode DISP'i'|MAIN|NONREC symbol.

The Janus RELEASE-statement is ignored.

LOC symbol.

The value of LC and symbol are stored in the symbol table.

```
  C1 LC = LC+1
"LC" CONTINUE
  C2 ID = IT(T1)-1
```

SPACE mode STATIC declarator [value].

The translation of static declarations is the same as the translation of the SPACE declarations within global declarations, except that no external symbol is defined. In other words, storage for these variables will also be allocated within the constant part of memory. The translation of this static declaration is the same as the translation of:
SPACE mode CSECT declarator [value].

Definitions.

DEF primitive-mode M identifier data-constant.

In the Janus program, identifiers can be associated with data-constants by manifest definitions. An applied occurrence of such an identifier is replaced by the data-constant. All these identifiers with the associated data-constants are stored in a compile-time manifest table and are accessible during the compilation of the whole Janus program.

SPACE mode CONST declarator value.

Depending on mode and value, storage is allocated within the constant part of memory: a row of DATA statements is generated, one for each primitive component of value. This value can be a data-constant (a character, integer or real value, or a compile-time expression), or an array-value or a record-value. The elements of the array and the fields of the record specify the DATA statements which are generated. symbol (of declarator) and the value of C are stored in the symbol table. It is worthwhile to note that global declarations, static declarations and constant definitions cause the same code to be produced.

ARRAY mode1 (PACK|ALIGN) mode2 ((expression)|[expression]).

All modes that are defined are recorded in the symbol table. Each mode is associated with an integer value, which indicates the number of storage units needed. As mentioned already, no distinction is made between packed and aligned arrays. The product of the value of expression and the value of mode1 is associated with mode2 and is stored in the symbol table.

RECORD mode1.

The first line of a Janus record-mode definition only serves to indicate that the next lines will contain a full specification of this mode. A compile-time counter CC is initialized to 0; this counter contains the number of storage units needed for a variable of mode1. When the RECEND-statement is encountered, the number of storage units effectively needed is assigned to mode1. Until that time the maximum storage to be reserved is stored in a compile-time variable MX. This variable is also initialized to zero. Two compile-time integer arrays are used; an array LM in which local maxima are stored and an array MS which is considered a compile-time stack to store the starting addresses of the variant-parts. The following code is generated:

```
Cl CC = 0
Cl MX = 0
```

FIELD mode2 (PACK|ALIGN) symbol [(expression)|[expression]].

symbol is recorded in the symbol table. The value of the counter CC (as mentioned in the description of RECORD) is associated with symbol. So symbol contains the displacement computed from the starting address of the record (as recorded in the variable involved). The counter CC is now increased by a number as designated by mode2.

```
Cl CC = CC + 'mode2' [* 'expression']
```

MARK 'i'.

This line begins a variant-part with at least one variant of level 'i'. As several variants occupy the same amount of storage it is necessary to reserve the largest amount of space needed by any specific variant of the given level. So it is necessary to save the current value of the counter CC to make it possible to pick up this value when the corresponding BACK line is encountered. These counter values must be saved on a compile-time stack since variants may be nested. This means that the following compile-time assignations take place:

```
Cl LM('i') = 0
Cl MS('i') = CC
```

BACK 'i' variant-id.

This Janus statement marks the end of a variant. I do not see any sense in the use of this variant-id symbol, since the Janus definition provides no meaning for it, but it may be that this symbol is used by Pascal J in the translation of a call to the Pascal 'new' function [3]. The total number of storage units reserved (as stored in the counter CC) will therefore be recorded in the symbol variant-id. This symbol is to be stored in the symbol table. If the first value is greater than MX, this value is assigned to MX. The value of the counter CC is compared with the value of MX. Moreover, the value stored the previous MARK line is assigned to the counter CC. I summarize all these compile-time actions in the following list:

```
C1 MX = MAX0(MX,CC)
C1 LM('i') = MAX0(LM('i'),CC)
C1 "variant-id" = LM('i')
C1 IF ('i'.GT.1) LM('i'-1) = MAX0(LM('i'),LM('i'-1))
C1 CC = MS('i')
```

RECEND model.

This Janus line closes a record definition. The maximum of the compile-time variable MX and the counter CC is associated with the symbol model, which is recorded in the symbol table.

RANGE INT'i' constant1 constant2.

As mentioned before, logical, real and integer entities occupy one storage unit in Fortran. Therefore, I ignore the constants. The mode INT'i' is equivalenced to the Janus mode INT. This interpretation is quite sensible, since range checking is not provided in Janus, and no machine-independent ANSI Fortran interpretation of the constants is possible. This specification could only be meaningful with a view to storage allocation.

PRECSN REAL'i' constant1 [constant2].

As I have mentioned before, I do not implement variables of type DOUBLE PRECISION, because they need two storage units in contrast with variables of type INTEGER, REAL or LOGICAL. Therefore, the interpretation of a PRECSN line is nearly the same as of a RANGE line. All types REAL'i' which are declared in the Janus program are equivalenced to the Janus mode REAL.

TRANSMISSION statements.

In this section Janus operations are discussed that move operands from and to the operand stack. These operations are composed of an operator and an optional operand. This operand is a data-constant, a reference to a constant, a variable or a temporary. If it is not a denotation, it is necessary to search the symbol, identifier or manifest

table for the address or value of the operand. The operand is moved from or to the top of the operand stack; the address of this top element is determined by two integers: MP, which points to the beginning of the current activation record (an execution-time variable), and a displacement expressed in a compile-time variable ID. The address of the top element of the operand stack is always denoted as: MP+"ID"; MP+"ID" points to the storage unit last occupied.
The push operations LOAD, SETLOC and TEST cause ID to be increased by 1 (C2 ID = ID+1); if, however, the LOAD-instruction is encountered and primitive-mode is PROC, ID is increased by 2; the pop operations BASE and INDEX cause ID to be decreased by 1 (C2 ID = ID-1); push operations cause ID to be changed before Fortran code is generated; pop instructions change the value of ID after code is generated. Generating code for these instructions is done during the first pass of compilation; however, during the first pass the addresses within the operand stack are not known since not all explicitly declared variables have already been seen; they are filled in during the second pass.
The CMPM-instruction sets the Janus condition-code. This instruction is implemented by storing the two operands in two character strings after the addresses of these operands are calculated. When the condition-code is tested (by conditional jump|trap or test instructions) these operands are used to compose a Fortran relational expression. This expression consists of the two operands and a relational operator that is derived from the condition-code occurring in the testing instruction.
The MOVE-instruction copies a mode object from one place in the Janus memory to another. A Janus mode is only used to determine the number of storage units to be copied, but not the type of them. I prefer a run-time routine that copies these storage units to a process that copies all these values as if they were of type real. Such a routine needs three parameters: the source address, the destination address and the number of storage units to be copied. In some sense the same holds for the CMPM-instruction; two mode objects are compared and the condition-code is set; the implementation can best be done by a run-time routine with three parameters.
The INDEX- and BASE-instructions may be provided with the (R)-flag. This flag indicates that not the top element, but the subtop element of the operand stack is the operand of the Janus operation. The type of the top element of the operand stack is unknown; my implementation does not allow this element to be a PROC-mode object, since these objects require two storage units instead of one; variables of type CHAR, INT, ADDR, BOOL and REAL occupy only one storage unit. This top element is moved to the position below the top. As the type of this element is unknown, the type of this variable is considered to be real. On most Fortran implementations the real assignment causes no problems.
The STORE-, CMPM- and MOVE-instructions may be provided with the (N)-flag. If so, ID is decreased by 1 (C2 ID = ID-1); if, however, the STORE-instruction is encountered and primitive-mode is PROC, ID is decreased by 2.
I will now give a detailed description for each Janus instruction; the following conventions are used:

MEM|BM|RM :	depending on the type of primitive-mode MEM, BM or RM is generated;
	type is INT, ADDR or CHAR : MEM

		type is BOOL : BM
		type is REAL : RM
		If type is PROC, code is generated on a different way.
AD	:	address of the variable, temporary or CONST reference.
OP	:	value of the operand of the operation.
MOVE,CMPM	:	run-time routines that implement the MOVE- and CMPM-instructions, respectively; they need 3 parameters: the source address, the destination address and the number of storage units to be copied or compared; the CMPM-routine delivers two character strings: "N" and "O"; in the execution-time variable N the value -1, 0 or 1 is stored indicating that the condition-code is set on LT, EQ or GT, respectively.
OPD1,OPD2	:	two character strings in which the operands are stored by a condition-code setting instruction.

LOAD primitive-mode operand.

```
        MEM|BM|RM(MP+"ID") = "OP"
```

If primitive-mode is PROC:

```
        MEM(MP+"ID-1") = MEM("AD")
        MEM(MP+"ID") = MEM("AD+1")
```

SETLOC ADDR data-location.

```
        MEM(MP+"ID") = "AD"
```

STORE[(N)] primitive-mode cell.

```
        MEM|BM|RM("AD") = MEM|BM|RM(MP+"ID")
```

If primitive-mode is PROC:

```
        MEM("AD+1") = MEM(MP+"ID")
        MEM("AD") = MEM(MP+"ID-1")
```

MOVE[(N)] mode variable.

```
        CALL MOVE(MEM(MP+"ID"),"AD",'mode')
```

CMPM[(N)] mode data-location.

```
        CALL CMPM(MEM(MP+"ID"),"AD",'mode')
```

TEST condition-code .

```
        BM(MP+"ID") = "OPD1"."condition-code"."OPD2"
```

TEST condition-code cell.

```
            BM("AD") = "OPD1"."condition-code"."OPD2"

    BASE ADDR .

            IBS = MEM(MP+"ID")

    BASE(R) ADDR .

            IBS = MEM(MP+"ID-1")
            RM(MP+"ID-1") = RM(MP+"ID")

    BASE ADDR operand.

            IBS = MEM("AD")

    BASLOC ADDR data-location.

            IBS = "AD"

    INDEX INT['i'] .

            IND = MEM(MP+"ID")

    INDEX(R) INT['i'] .

            IND = MEM(MP+"ID-1")
            RM(MP+"ID-1") = RM(MP+"ID")

    INDEX INT['i'] operand.

            IND = "OP"
```

COMPUTATIONAL statements.

All Janus computational instructions consist of an operator and an optional operand. The operator is monadic or dyadic. If no operand is given it is assumed to be on the operand stack. The left operand for a dyadic operator is always on the operand stack; if operand is present it is the right one; otherwise the right operand is the top element of the operand stack and the left operand is the subtop element.

Each operator delivers a value; the type of this value depends on the operator and the type of the operands; the resulting value is pushed onto the operand stack. Monadic operators - ABS, NEG, NOT, TRUNCN and TRUNCZ - cause ID to be increased by 1 (C2 ID = ID+1) if operand is present; otherwise ID remains unchanged (the operand is overwritten). Dyadic operators - ADD, AND, CMP, DIV, EQSGN, EXOR, MOD, MPY, OR and SUB - cause ID to be decreased by 1 if no operand is given; otherwise ID remains unchanged; in both cases the left operand is overwritten.

The CMP- and EQSGN-instructions set the condition-code; at the translation of the EQSGN-instruction two new operands are created; these two operands make it possible this instruction to be implemented in the same way as the other condition-code setting instructions.

The DIV-, SUB- and MOD-instructions may be provided with the (R)-flag. If so, the operands are exchanged.

The EQSGN- and CMP-instructions may be provided with the (N)-flag. If so, ID is decreased by 2 if no operand is given; otherwise ID is decreased by 1.

The conventions used are the same as in the previous section. Below, I give the code for the instructions in case no operand is present. The implementation of the instructions provided with operand is nearly the same.

```
ABS INT|REAL .

        MEM|RM(MP+"ID") = IABS|ABS(MEM|RM(MP+"ID"))

ADD INT|REAL .

        MEM|RM(MP+"ID-1") = MEM|RM(MP+"ID-1") + MEM|RM(MP+"ID")

AND BOOL .

        BM(MP+"ID-1") = BM(MP+"ID-1") .AND. BM(MP+"ID")

CMP[(N)] INT|REAL .

     C1 OPD1 = MEM|RM(MP+"ID-1")
     C1 OPD2 = MEM|RM(MP+"ID")

DIV INT|REAL .

        MEM|RM(MP+"ID-1") = MEM|RM(MP+"ID-1") / MEM|RM(MP+"ID")

DIV(R) INT|REAL .

        MEM|RM(MP+"ID-1") = MEM|RM(MP+"ID") / MEM|RM(MP+"ID-1")

EQSGN[(N)] INT|REAL .

     C1 OPD1 = ISIGN|SIGN(MEM|RM(MP+"ID"),MEM|RM(MP+"ID-1"))
     C1 OPD2 = MEM|RM(MP+"ID")

EXOR BOOL .

        BM(MP+"ID-1") = (BM(MP+"ID-1").OR.BM(MP+"ID")) .AND.
                        (.NOT.BM(MP+"ID-1").OR..NOT.BM(MP+"ID"))

MPY INT|REAL .

        MEM|RM(MP+"ID-1") = MEM|RM(MP+"ID-1") * MEM|RM(MP+"ID")

NEG INT|REAL .
```

```
          MEM|RM(MP+"ID") = - MEM|RM(MP+"ID")

  NOT BOOL .

          BM(MP+"ID") = .NOT. BM(MP+"ID")

  OR BOOL .

          BM(MP+"ID-1") = BM(MP+"ID-1") .OR. BM(MP+"ID")

  SUB INT|REAL .

          MEM|RM(MP+"ID-1") = MEM|RM(MP+"ID-1") - MEM|RM(MP+"ID")

  SUB(R) INT|REAL .

          MEM|RM(MP+"ID-1") = MEM|RM(MP+"ID") - MEM|RM(MP+"ID-1")

  TRUNCN INT .

          X = RM(MP+"ID")
          MEM(MP+"ID") = INT(X)
          IF (FLOAT(MEM(MP+"ID")).GT.X) MEM(MP+"ID") = MEM(MP+"ID") - 1

  TRUNCZ INT .

          MEM(MP+"ID") = INT(RM(MP+"ID"))
```

OPTIONAL statements.

The operations I have discussed until now are available in every implementation of Janus. Furthermore, some operators are "optional", that is to say they are not included within Standard Janus. All operators in this section except LDEX are monadic. The operator ODD that sets the condition-code may be provided with the (N)-flag; if so, ID is decreased by 1. In the same way as the EQSGN-instruction, the ODD-instruction delivers two operands in the character strings OPD1 and OPD2. The EXPO- and LDEX-instructions handle the exponent of an operand. It is not easy to isolate the exponent of a real variable in Fortran, nor is it clear what it would mean in Standard Fortran; it is not even clear what it would mean, since Janus has no way of indicating the base. For these reasons I do not implement these instructions.

```
  ODD[(N)] INT .

        C1 OPD1 = IABS(MOD(MEM(MP+"ID"),2))
        C1 OPD2 = 1

  ROUND INT|REAL .
```

```
        MEM|RM(MP+"ID") = INT|AINT(RM(MP+"ID")+SIGN(0.5,RM(MP+"ID")))

SIGN INT|REAL .

        IF (MEM|RM(MP+"ID").NE.0|0.)
       CMEM(MP+"ID") = ISIGN|SIGN(1|1.,MEM|RM(MP+"ID"))
        IF (RM(MP+"ID").EQ.0.) MEM(MP+"ID") = 0

SQUARE INT|REAL .

        MEM|RM(MP+"ID") = MEM|RM(MP+"ID") * MEM|RM(MP+"ID")

TRUNCN REAL .

        X = RM(MP+"ID")
        RM(MP+"ID") = AINT(X)
        IF (RM(MP+"ID").GT.X) RM(MP+"ID") = RM(MP+"ID") - 1.0

TRUNCZ REAL .

        RM(MP+"ID") = AINT(RM(MP+"ID"))
```

<u>Conversions</u>.

In Janus it is possible to convert a value of some type into a value of other types. Some of these conversions are defined by Janus, but the conversion of others is left to the implementer. Janus includes 9 conversion operators. Since I do not implement subranges of the modes REAL and INT, the conversions REAL REAL and INT INT are implemented as if they were copying instructions. As mentioned already, Janus characters are implemented as integer values using two transform functions. This implies that the conversions CHAR INT and INT CHAR are also implemented as if they were copying instructions. The implementation of the conversions BOOL INT, INT BOOL, REAL INT and INT REAL is trivial. Discussion about the conversion PROC DISP'i' is held separately.

```
BOOL INT .

        BM(MP+"ID") = MEM(MP+"ID").EQ.1

INT BOOL .

        I = 0
        IF (BM(MP+"ID")) I = 1
        MEM(MP+"ID") = I

INT REAL .

        MEM(MP+"ID") = RM(MP+"ID")

REAL INT .
```

```
        RM(MP+"ID") = MEM(MP+"ID").
```

PROC DISP'i' _operand_.

This Janus instruction converts an ADDR-mode object into a PROC-mode object. The creation of a PROC-mode object takes place in two cases: 1. _operand_ denotes a Janus procedure or 2. _operand_ denotes a Janus label. My implementation requires an explicit _operand_, since otherwise no distinction can be made between operands that denote Janus procedures and those that denote Janus labels; if _operand_ denotes a Janus procedure, _symbol_ (of _operand_) is in the procedure table PT. PROC-mode objects are created for labels whose applied occurrence may be within an other procedure (i.e. a jump out of a procedure). To make a treatment similar to that of a procedure possible, the display level of the label has to be one more than the display level of the procedure in which this label is contained. In both cases - a call to the procedure or a jump to the label - the result is a call of the Fortran subroutine corresponding to a Janus procedure. It is not sufficient to specify which procedure is meant; the addressing environment (as indicated by DISP'i') must also be specified. Clearly, this instruction is contained within a Janus procedure of level j (DISP'j'). It is necessary that i $\leq$ j+1. MP marks the beginning of the activation record of the subroutine (procedure) currently active. The code that is generated depends on the value of j-i+1. The number of the procedure (if _operand_ denotes a Janus procedure) or the entry-point corresponding to the label (if _operand_ denotes a label) is pushed onto the operand stack.

```
       C2 ID = ID+1

if i = j+1             : MEM(MP+"ID") = MP

if i = j               : MEM(MP+"ID") = MEM(MP)

if i = j-1             : K = MEM(MP)
                         MEM(MP+"ID") = MEM(K)

otherwise(i ≤ j-2)     : K = MEM(MP)
                         L = "DL"-'i'-1
                      C1 LC = LC+1
                         DO "LC" M = 1,L
                    "LC" K = MEM(K)
                         MEM(MP+"ID") = MEM(K)

       C2 ID = ID+1
```

If _operand_ denotes a procedure:

```
          MEM(MP+"ID") = 'symbol'
```

If _operand_ denotes a label:

```
       C1 LA(T1) = LA(T1)+1
       C1 EP(T1,LA(T1)) = 'symbol'
```

```
        MEM(MP+"ID") = "LA(T1)"
```

ALLOCATION statements.

When a block of storage is claimed or freed the run-time routines GRAB and FREE are used. Code for these routines is given in APPENDIX 2. No distinction is made between STACK- and HEAP-instructions.

GRAB STACK|HEAP index.

```
     C2 ID = ID+1
        MEM(MP+"ID") = GRAB('index')
```

FREE STACK|HEAP index.

```
        CALL FREE(IBS,'index')
```

CONTROL statements.

JMP target.

If target is a code-location, this instruction is interpreted as a jump within the current procedure. In my implementation code-location is allowed to be only of the form: R symbol. The value that is associated with symbol is filled in during the second pass; in this way there are no problems with forward references of labels.

```
        GO TO 'symbol'
```

If target is a cell, this cell is considered to be a PROC-mode object and the instruction is interpreted as a jump out of the current procedure.

```
        MP = MEM("AD")
        MEM(MP+3) = MEM("AD+1")
        RETURN
```

JMP condition-code target.

If target is a code-location:

```
        IF ("OPD1"."condition-code"."OPD2") GOTO 'symbol'
```

If target is a cell:

```
     C1 LC = LC+1
        IF (.NOT.("OPD1"."condition-code"."OPD2")) GOTO "LC"
        MP = MEM("AD")
        MEM(MP+3) = MEM("AD+1")
        RETURN
     "LC" CONTINUE
```

TRAP [condition-code] identifier.

The implementation of the Janus TRAP-instruction can best be left to the implementer on each machine, because he knows all about the peculiarities of his machine. If a trap occurs, further execution of a Janus program is undefined; Janus does not define the meanings of any trap identifier. By IDF, I denote a traproutine whose specification is given by each individual implementer.

If condition-code is absent:

```
          CALL "IDF"
```

and otherwise:

```
          IF ("OPD1"."condition-code"."OPD2") CALL "IDF"
```

BEGCAS[(N)] primitive-mode .

It is impossible to select on a PROC-mode, an ADDR-mode or a BOOL-mode object, since there are no data-constants of these modes.

If primitive-mode is INT or CHAR:

```
          I = MEM(MP+"ID")
```

If primitive-mode is REAL:

```
          X = RM(MP+"ID")
```

If (N) is present:

```
       C2 ID = ID-1
```

CASE primitive-mode code-location data-constant.

DC denotes a Fortran integer or real constant equivalent to the Janus data-constant.

```
          IF (I|X.EQ."DC") GOTO 'symbol'
```

I admit that this translation may very often not be the most efficient one. For example, if it concerns an integer case-selection and the data-constants start from one with steps of one, it is obvious preferable to generate a Fortran COMPUTED GOTO statement. However, this requires a lot of administration since all data-constants and code-locations must be stored until the ENDCAS-statement is encountered. At that moment it is possible to make a choice between a COMPUTED GOTO statement and a row of LOGICAL IF statements. Furthermore, the values of the data-constants may be machine-independent. At this moment I have not chosen such a solution, although I realize that in most implementations such an analysis may be necessary to achieve sufficient efficiency.

ENDCAS primitive-mode [code-location].

If code-location is absent, no code is generated;

otherwise:

```
     GOTO 'symbol'
```

[R]CALL [primitive-mode] target.

As already mentioned, no distinction is made between recursive and nonrecursive calls. Nonrecursive calls are treated as if they were recursive. No code is generated.

[R]ARG[(N)] primitive-mode [operand].

In my implementation, the arguments for a call are pushed onto the operand stack. If operand is absent, the argument is already on the operand stack and no action is performed; otherwise this instruction is interpreted as a LOAD-instruction. I pay no attention to the occurrence of (N), since all arguments must be left on the operand stack. The arguments are assumed to be of primitive-mode; if the mode of an argument is not restricted, a run-time routine is used to copy operand to the operand stack (see page 19: the MOVE-instruction).

```
C2 ID = ID+1
   MEM|BM|RM(MP+"ID") = "OP"
```

If primitive-mode is PROC:

```
C2 ID = ID+2
   MEM(MP+"ID-1") = MEM("AD")
   MEM(MP+"ID") = MEM("AD+1")
```

[R]ARGLOC[(N)] ADDR data-location.

```
C2 ID = ID+1
   MEM(MP+"ID") = "AD"
```

[R]CEND [primitive-mode] target.

If target is a code-location:

```
   MEM(MP+"ID+1") = MP
   MEM(MP+"ID+2") = 'symbol'
```

If target is a cell:

```
   MEM(MP+"ID+1") = MEM("AD")
```

```
        MEM(MP+"ID+2") = MEM("AD+1")
```

In both cases:

```
        MEM(MP+"ID+3") = MP
        MEM(MP+"ID+4") = 1
     C1 LA(T1) = LA(T1) + 1
     C1 LC = LC+1
     C1 EP(T1,LA(T1)) = LC
        MEM(MP+3) = "LA(T1)"
        MP = MP+"ID+1"
        RETURN
   "LC" CONTINUE
     C2 ID = ID - TP('symbol'|MEM("AD+1")) [+ 'primitive-mode']
```

RETURN [primitive-mode] procedure-class symbol.

If primitive-mode is present, a value of this mode is to be placed on the top of the operand stack.

```
        MEM|BM|RM(MP-"TP(T1)") = MEM|BM|RM(MP+"ID")
```

If it is a PROC value:

```
        MEM(MP-"TP(T1)") = MEM(MP+"ID-1")
        MEM(MP-"TP(T1)-1") = MEM(MP+"ID")
```

Furthermore - also if primitive-mode is absent - :

```
        MP = MEM(MP+2)
        RETURN
```

STOP [primitive-mode] procedure-class symbol.

```
        STOP
```

ABORT [primitive-mode] procedure-class symbol.

```
        STOP
```

OMISSIONS AND PROBLEMS

The implementation of several Janus concepts is not given or is left to each individual implementer. A summary is given now of all points in the Janus definition that are not discussed.

1. Code-locations are allowed to be only of the form: R symbol.
2. The INDEX- and BASE-instructions cause problems if the (R)-flag is present.
3. The implementation of the TRAP-instruction is left to each individual implementer.
4. No subranges (RANGE) and precisions (PRECSN) are implemented.

5. EXPO and LDEX are not implemented.
6. No difference is made between aligned and packed arrays.

FINAL REMARKS

In this paper only a sketch has been given of a portable implementation. I have tried not to deviate from the Janus document, but at some places I have chosen for an interpretation which seems the best possible to me. I have done so only where the Janus document is not clear or does not specify exactly what is meant. Janus includes some less successful concepts, which are hopefully replaced in a later version; the current language is not very suitable to be used as intermediate code for the translation of high-level languages. I hope that this paper may contribute to a real implementation and may remove some difficulties which might otherwise appear.

ACKNOWLEDGEMENT

I am very grateful to H.J. Boom for his helpful advice and his patient explanations which were essential for the completion of this paper.

REFERENCES

-[1] Waite, W.M. and B.K. Haddon, A preliminary definition of Janus, Report SEG-75-1, Department of Electrical Engineering, University of Colorado.
-[2] Standard FORTRAN Programming Manual, the National Computing Centre Limited, 1972, SBN 85012 063 2.
-[3] Jensen, Kathleen and Niklaus Wirth, PASCAL User Manual and Report, Springer Verlag, 1975, ISBN 0-387-90144-2.

APPENDIX 1

It is not easy to implement Janus composed values in a correct and efficient way. In this section an algorithm is given, described in Algol 68. This algorithm may be a guide for an implementer to write his own version. The following variables are used:

a1,a2,a3:	one-dimensional integer arrays; a1 and a2 contain the values of _mode_ and _expression_, respectively; a3 is used to store the nesting level upon encountering an ARVND-line.
c	: constant counter.
n	: number of times a primitive value (data-constant) is put out.
t	: nesting level.
k	: index in array a3.

The constant definitions etc. are implemented as DATA statements that are generated on a separate constant file; for this constant file I have chosen the file stand out (standard output file).
When the first ELMV- or FLDV-line is encountered, the integer variables n, t and k have been initialized by the previous SPACE-line:

```
n := 0; t := 1; k := 0;
```

The composed value is read from the Janus source file (standard input file) by:

```
while t/=0 do readline((instruction,newline)) od;
```

The instruction can be:

1. ELMV-instruction
2. FLDV-instruction
3. RCVND-instruction
4. ARVND-instruction

The _value_ on the end of the ELMV-, the FLDV- or the ARVND-line may be a data-constant. I assume this data-constant is a real denotation. It is not difficult to see what to do when another kind of data-constant occurs. The data-constant is stored in a _string_ s;

ELMV _mode_ CONST|CSECT|STATIC [_expression_]|(_expression_) [_value_].

```
a1[t] := 'mode';
a2[t] := 'expression';
```

a. _value_ is absent:

```
c := c + a1[t] * a2[t];
```

b. _value_ is _data-constant_:

```
print("DATA ");
p(1);
if s[1]="-"
```

```
   then s := s[1] + "." + s[2: ]
   else s := "." + s
   fi;
   print(("/",if n>1 then (whole(n,0),"*") else "" fi,
          s,"/",newline));
   c := c + a2[t];
   n := 0;
```

c. value begins with a "+":

```
   t := t + 1;
```

FLDV mode CONST|CSECT|STATIC symbol [[expression]|(expression)] [value].

```
   a1[t] := 'mode';
   a2[t] := 'expression' (if absent then 1);
```

a. value is absent:

```
   c := c + a1[t] * a2[t];
```

b. value is data-constant:

```
   print("DATA ");
   p(1);
   if s[1]="-"
   then s := s[1] + "." + s[2: ]
   else s := "." + s
   fi;
   print(("/",if n>1 then (whole(n,0),"*") else "" fi,
          s,"/",newline));
   c := c + a2[t];
   n := 0;
```

c. value begins with a "+":

```
   a2[t] := 1;
   t := t + 1;
```

RCVND mode CONST|CSECT|STATIC [symbol].

```
   while t := t - 1;
         if t>0 then c := c + a1[t] * (a2[t] - 1) fi;
         if k=0 then false else a3[k]=t fi
   do k := k - 1 od;
```

ARVND mode CONST|CSECT|STATIC [symbol] [expression]|(expression) [value].

```
   a1[t] := 'mode';
   a2[t] := 'expression';
```

a. value is absent:

```
   c := c + a1[t] * a2[t];
   while t := t - 1;
         if t>0 then c := c + a1[t] * (a2[t] - 1) fi;
         if k=0 then false else a3[k]=t fi
   do k := k - 1 od;
```

b. value is data-constant:

```
   print("DATA ");
   p(1);
   if s[1]="-"
   then s := s[1] + "." + s[2: ]
   else s := "." + s
   fi;
   print(("/",if n>1 then (whole(n,0),"*") else "" fi,
          s,"/",newline));
   c := c + a2[t];
   n := 0;
   while t := t - 1;
         if t>0 then c := c + a1[t] * (a2[t] - 1) fi;
         if k=0 then false else a3[k]=t fi
   do k := k - 1 od;
```

c. value begins with a "+":

```
   k := k + 1;
   a3[k] := t;
   t := t + 1;
```

The code of the procedure p:

```
   proc p= (int i) void:
        (if i=t
         then print((if n>0 then "," else "" fi,"RM(",
                    whole(c,0),")"));
              n := n + 1;
              to a2[t] - 1
              do c := c + a1[t]; n := n + 1;
                 print((",RM(",whole(c,0),")"))
              od
         else p(i+1);
              to a2[i] - 1
              do c := c + a1[i]; p(i+1) od
         fi;
         c := c - a1[i] * (a2[i] - 1)
        );
```

APPENDIX 2

By the routines GRAB and FREE a block of storage is dynamically claimed or freed, respectively.

```
      INTEGER FUNCTION GRAB(N)
      COMMON/H/ MP,MEM("SIZE")
      K = "SIZE"
   10 M = MEM(K)
      IF (MEM(M).EQ.0) GOTO 40
      IF (MEM(M-1).EQ.N) GOTO 30
      IF (MEM(M-1).GE.N+2) GOTO 20
      K = M
      GOTO 10
   20 L = M-N
      MEM(K) = L
      MEM(L) = MEM(M)
      MEM(L-1) = MEM(M-1) - N
      GRAB = M
      RETURN
   30 MEM(K) = MEM(M)
      GRAB = M
      RETURN
   40 L = M-N
      MEM(K) = L
      MEM(L) = 0
      GRAB = M
      RETURN
      END

      SUBROUTINE FREE(N1,N2)
      COMMON/H/ MP,MEM("SIZE")
      K = "SIZE"
   10 M = MEM(K)
      IF (M.LT.N1) GOTO 20
      K = M
      GOTO 10
   20 J = N1-N2
      IF (K.EQ."SIZE") GOTO 40
      I = K - MEM(K-1)
      IF (I.NE.N1) GOTO 40
      IF (J.NE.M) GOTO 30
      MEM(K) = MEM(M)
      IF (MEM(M).NE.0) MEM(K-1) = MEM(K-1) + N2 + MEM(M-1)
      RETURN
   30 MEM(K-1) = MEM(K-1) + N2
      RETURN
   40 IF (J.NE.M) GOTO 50
      MEM(K) = N1
      MEM(N1) = MEM(M)
      IF (MEM(M).NE.0) MEM(N1-1) = MEM(M-1) + N2
      RETURN
   50 MEM(K) = N1
```

```
      MEM(N1) = M
      MEM(N1-1) = N2
      RETURN
      END
```

APPENDIX 3

As an example a Janus program is given and the translation of this program into Standard Fortran. In the left column the Janus program and in the right column comment is given in Algol 68 style. The program concerns the computation of the Ackermann function.

```
START ACKERMANN.
BEGIN MAIN A0.                  begin
PAREND MAIN A0.
SPACE PROC DISP0 A1.
PROC DISP1 R T1.
STORE(N) PROC LOCAL A1.         proc ack = ...
RCALL INT DISP0 A1.             ack
RARG(N) INT AINT 3.             (3
RARG(N) INT AINT 4.             ,4
RCEND INT DISP0 A1.             )
RETURN MAIN A0.
END MAIN A0.                    end
BEGIN INT DISP1 T1.
PARAM INT DISP1 M1.             (m
PARAM INT DISP1 N1.             ,n
PAREND INT DISP1 T1.            )
LOAD INT LOCAL M1.              if m
CMP(N) INT AINT 0.              = 0
JMP NE R L0.
LOAD INT LOCAL N1.              then n
ADD INT AINT 1.                 + 1
RETURN INT DISP1 T1.
LOC L0.                         elif
LOAD INT LOCAL N1.              n
CMP(N) INT AINT 0.              = 0
JMP NE R L1.
RCALL INT DISP0 A1.             then ack(
LOAD INT LOCAL M1.              m
SUB INT AINT 1.                 -1
RARG(N) INT.
RARG(N) INT AINT 1.             ,1
RCEND INT DISP0 A1.             )
RETURN INT DISP1 T1.
LOC L1.                         else
RCALL INT DISP0 A1.             ack(
LOAD INT LOCAL M1.              m
SUB INT AINT 1.                 -1
RARG(N) INT.
RCALL INT DISP0 A1.             ,ack(
RARG(N) INT LOCAL M1.           m
LOAD INT LOCAL N1.              ,n
SUB INT AINT 1.                 -1
RARG(N) INT.
RCEND INT DISP0 A1.             )
RARG(N) INT.
RCEND INT DISP0 A1.             )
```

```
  RETURN INT DISP1 T1.
  END INT DISP1 T1.                fi
  FINISH.
```

In the Fortran program the statements which are the translation of a Janus instruction are preceded by that Janus instruction as a comment line. The program causes 125 to be printed.

```
C  START ACKERMANN.
C  BEGIN MAIN A0.
      SUBROUTINE P2
C  PAREND MAIN A0.
      LOGICAL BM(1000)
      REAL RM(1000)
      COMMON/H/ MP,MEM(1000)
      EQUIVALENCE (MEM(1),BM(1),RM(1))
      I=MEM(MP+3)
      GO TO (1,2),I
    1 CONTINUE
C  SPACE PROC DISP0 A1.
C  PROC DISP1 R T1.
      MEM(MP+6)=MP
      MEM(MP+7)=3
C  STORE(N) PROC LOCAL A1.
      MEM(MP+5)=MEM(MP+7)
      MEM(MP+4)=MEM(MP+6)
C  RCALL INT DISP0 A1.
C  RARG(N) INT AINT 3.
      MEM(MP+6)=3
C  RARG(N) INT AINT 4.
      MEM(MP+7)=4
C  RCEND INT DISP0 A1.
      MEM(MP+8)=MEM(MP+4)
      MEM(MP+9)=MEM(MP+5)
      MEM(MP+10)=MP
      MEM(MP+11)=1
      MEM(MP+3)=2
      MP=MP+8
      RETURN
    2 CONTINUE
C  PRINT RESULT (NO JANUS INSTRUCTION)
      WRITE(6,30) MEM(MP+6)
   30 FORMAT(I8)
C  RETURN MAIN A0.
      MP=MEM(MP+2)
      RETURN
C  END MAIN A0.
      END
C  BEGIN INT DISP1 T1.
      SUBROUTINE P3
C  PARAM INT DISP1 M1.
C  PARAM INT DISP1 N1.
C  PAREND INT DISP1 T1.
```

```
      LOGICAL BM(1000)
      REAL RM(1000)
      COMMON/H/ MP,MEM(1000)
      EQUIVALENCE(MEM(1),BM(1),RM(1))
      I=MEM(MP+3)
      GO TO (1,3,5,6),I
    1 CONTINUE
C   LOAD INT LOCAL M1.
      MEM(MP+4)=MEM(MP-2)
C   CMP(N) INT AINT 0.
C   JMP NE R L0.
      IF (MEM(MP+4).NE.0) GO TO 2
C   LOAD INT LOCAL N1.
      MEM(MP+4)=MEM(MP-1)
C   ADD INT AINT 1.
      MEM(MP+4)=MEM(MP+4)+1
C   RETURN INT DISP1 T1.
      MEM(MP-2)=MEM(MP+4)
      MP=MEM(MP+2)
      RETURN
C   LOC L0.
    2 CONTINUE
C   LOAD INT LOCAL N1.
      MEM(MP+4)=MEM(MP-1)
C   CMP(N) INT AINT 0.
C   JMP NE R L1.
      IF (MEM(MP+4).NE.0) GO TO 4
C   RCALL INT DISP0 A1.
C   LOAD INT LOCAL M1.
      MEM(MP+4)=MEM(MP-2)
C   SUB INT AINT 1.
      MEM(MP+4)=MEM(MP+4)-1
C   RARG(N) INT.
C   RARG(N) INT AINT 1.
      MEM(MP+5)=1
C   RCEND INT DISP0 A1.
      I=MEM(MP)
      MEM(MP+6)=MEM(I+4)
      MEM(MP+7)=MEM(I+5)
      MEM(MP+8)=MP
      MEM(MP+9)=1
      MEM(MP+3)=2
      MP=MP+6
      RETURN
    3 CONTINUE
C   RETURN INT DISP1 T1.
      MEM(MP-2)=MEM(MP+4)
      MP=MEM(MP+2)
      RETURN
C   LOC L1.
    4 CONTINUE
C   RCALL INT DISP0 A1.
C   LOAD INT LOCAL M1.
```

```
      MEM(MP+4)=MEM(MP-2)
C   SUB INT AINT 1.
      MEM(MP+4)=MEM(MP+4)-1
C   RARG(N) INT.
C   RCALL INT DISPO A1.
C   RARG(N) INT LOCAL M1.
      MEM(MP+5)=MEM(MP-2)
C   LOAD INT LOCAL N1.
      MEM(MP+6)=MEM(MP-1)
C   SUB INT AINT 1.
      MEM(MP+6)=MEM(MP+6)-1
C   RARG(N) INT.
C   RCEND INT DISPO A1.
      I=MEM(MP)
      MEM(MP+7)=MEM(I+4)
      MEM(MP+8)=MEM(I+5)
      MEM(MP+9)=MP
      MEM(MP+10)=1
      MEM(MP+3)=3
      MP=MP+7
      RETURN
    5 CONTINUE
C   RARG(N) INT.
C   RCEND INT DISPO A1.
      I=MEM(MP)
      MEM(MP+6)=MEM(I+4)
      MEM(MP+7)=MEM(I+5)
      MEM(MP+8)=MP
      MEM(MP+9)=1
      MEM(MP+3)=4
      MP=MP+6
      RETURN
    6 CONTINUE
C   RETURN INT DISP1 T1.
      MEM(MP-2)=MEM(MP+4)
      MP=MEM(MP+2)
      RETURN
C   END INT DISP1 T1.
      END
C   FINISH .
      PROGRAM SPV(OUTPUT,TAPE6=OUTPUT)
      LOGICAL BM(1000)
      REAL RM(1000)
      COMMON/H/ MP,MEM(1000)
      EQUIVALENCE (MEM(1),BM(1),RM(1))
      MEM(MP-3)=1
      MEM(MP)=MP-4
      MEM(MP+1)=2
      MEM(MP+2)=MP-4
      MEM(MP+3)=1
    1 I=MEM(MP+1)
      GO TO (11,12,13),I
   12 CALL P2
   GO TO 1
13 CALL P3
   GO TO 1
11 STOP
   END
   BLOCK DATA
   LOGICAL BM(1000)
   REAL RM(1000)
   COMMON/H/ MP,MEM(1000)
   EQUIVALENCE (MEM(1),BM(1),RM(1))
   DATA MP/5/
   DATA MEM(1000)/999/,MEM(999)/0/
   END
```