SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 4, 5–23 (1974)

The Mobile Programming System, Janus

S. S. COLEMAN, P. C. POOLE AND W. M. WAITE
Department of Electrical Engineering, University of Colorado, Boulder, Colorado 80302, U.S.A.

SUMMARY

Janus is a symbolic language used to embody the information which is normally passed from the analysis phase of a compiler to the code generators. It is designed for transporting software: A program coded in a high level language can be translated to Janus on one computer, and the resulting output translated to assembly code on another. (The STAGE2 macro processor could be used for the second translation.) In this paper we present the principles upon which Janus is based, and show that it is suited to a wide range of source languages and target computers.

KEY WORDS Portability Intermediate language UNCOL Abstract machine model

INTRODUCTION

The Mobile Programming System[1–4] is a collection of highly portable software which can be implemented by means of a full bootstrap.[5,6] Each routine is expressed as a program for a ficticious computer known as an *abstract machine model.* These programs can be translated to assembly code programs for the target computer by a macro processor.[5,7,8] To start the bootstrap, a simple macro processor (equivalent to approximately 80 lines of FORTRAN code[1]) must be implemented on the target machine. Similar systems have been described by other authors.[9–11]

A previous paper[6] gave a critical evaluation of the abstract machine modelling technique and of three particular models. The main conclusion reached was that, while the technique has been demonstrated to be viable for constructing portable software, major problems are associated with the design of a suitable abstract machine model (and its programming language) for a given application. The task is not easy even for experienced programmers. Further, the diversity of designs led to implementations which could not make use of earlier realizations even though the models had many features in common. The effort required to move software was therefore greater than it needed to be.

Further standardization appeared to be the key to our difficulties, so we undertook an examination of existing programming languages, translators and machines to determine a set of common structural attributes. The results of this study, summarized in the next section, led to the design of an extensible intermediate language. We named this language *Janus*, after the Roman deity who was the patron of gates and doors. An intermediate language is a gate, through which an algorithm must pass to reach the machine. We found that our major design problem was to determine the ‘width’ of the gate: How much information about the source program must be carried in the intermediate code to permit efficient implementation on a variety of hardware.

It is well known that the translation of a programming language consists of two steps—analysis and code generation. In the next section, we will suggest separating these steps.

Received 25 April 1973
Revised 20 August 1973

Indeed, compilers have been designed[12] to use a common body of procedures for code generation: there is one procedure for each object statement type, and these are called by the analysis routines with vectors describing the particular instance of the statement to be generated. We conceive of Janus as a symbolic representation of these calls. By expressing this information in symbolic form, we can extract the algorithm and implement it on another machine simply by providing code generation procedures.

Interpretive versions of the code generators can be written in the form of macros which express constructs of the intermediate language in terms of the assembly code of the target computer. These macros are interpreted by a macro processor, and 'called' by the lines of symbolic Janus code. We shall show the form of these symbolic lines and explain the information which they contain. We shall also indicate how the code generation macros might be implemented on the major classes of computers.

It will be clear to the reader even at this point that there is an UNCOL-like flavour about the design of Janus. We shall compare Janus with similar projects such as UNCOL[13-16] and SLANG.[17] We should perhaps stress that each of these names actually denotes a family of abstract machines. All machines in each family have a common basic structure, and a common symbolic language is used to program them. The set of instructions may, however, vary considerably from one member of the family to another. As Sibley points out:[17]

> 'No stigma will be attached to such (variations in the instruction set) since the techniques used for processing the (symbolic instructions) are relatively independent of any particular set of them.'

It is this variability of instruction set which allows one to use the same intermediate language as a means for encoding programs to solve many different problems.

THE ORGANIZATION OF JANUS

Every programming language has three components.

(1) A set of *primitive modes*: classes of data items which are considered atomic in the language. Examples are integers, reals and Booleans.

(2) A set of *primitive operators*: commands which transform data items and are not themselves defined in terms of other operators. Examples are integer subtraction, real multiplication and the relational operator 'less than' with real operands.

(3) A set of *formation rules*: mechanisms which allow a user to construct new modes and operators from existing ones. Examples are the mechanisms for defining arrays and procedures.

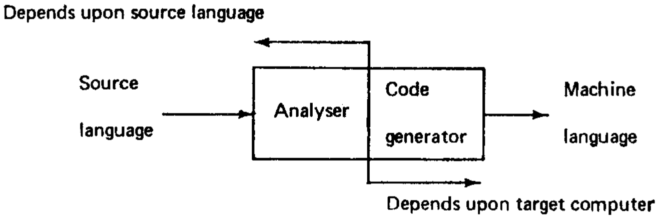
The particular sets of primitive modes and operators provided by a programming language are determined by the class of problems for which the language is designed. The set of formation rules, on the other hand, seems to be largely independent of problem class.[18] Table I summarizes the formation rules normally found in 'high-level' languages.

When a language is translated, the primitive modes and operators appear essentially unchanged in the machine code. Primitive operators may be realized by single instructions, instruction sequences or subroutine calls; nevertheless, there is a one-to-one correspondence between the appearance of the operator in the source code and the appearance of its realization in the machine code. In contrast, many of the formation rules are dealt with by the translator, and appear in a drastically altered form (if they appear at all) in the machine code. This alteration of the formation rules is one of the primary purposes of the translator, and it is dependent upon the source language rather than the target machine. Thus we may partition a translator into two sections, as shown in Figure 1(a). The analyser which deals with the formation rules depends upon the source language, while the code generator depends upon the target computer.

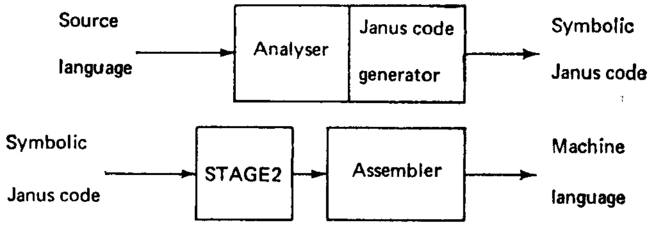
Table I. Formation rules

Operator creation	Mode creation
Expression	Array
Conditional (and case)	Structure (record)
Iteration	Reference (pointer)
Procedure	Domain restriction
Coroutine	Enumeration

It is clear that the major flow of information across the interface in Figure 1(a) is from left to right. This information consists primarily of operator/operand pairs which specify the sequence of actions embodied in the source program.[13]



(a) A conventional translator



(b) Janus as an intermediate language

Figure 1. The translation process

In conventional translators there is also some flow of information from right to left across the interface. Our study showed that this information was related to the fact that some of the formation rules of Table I are at least partially reflected in the architecture of real computers. We can therefore compensate for our lack of right-to-left flow by including additional information in the Janus instructions about certain formation rules.

Figure 1(b) illustrates the role of Janus in the translation process: The translator is split along the interface of Figure 1(a). A small module attached to the analyser encodes the information normally passed across the interface from left to right. The encoded information constitutes a symbolic program, which can then be transmitted to another computer. Janus specifies the structure of this symbolic program, but says nothing about the particular set of operators and modes which can be used.

The symbolic Janus code is translated to the assembly language of the target computer by a program such as STAGE2.[4] Simple translation rules are supplied by the user to describe the various Janus constructs and the primitive modes and operators. Final translation to objects code is provided by the normal assembler of the target computer.

The Janus architecture

Our design for the Janus family of abstract machines is based upon the relationship between existing languages and existing hardware. Each component of Figure 2 models a specific language characteristic; the precise form of the model was chosen to simplify the generation of machine code from symbolic Janus. (The code generator should be simplified at the expense of the analyser because the former must be respecified for each machine while the latter remains unchanged.)

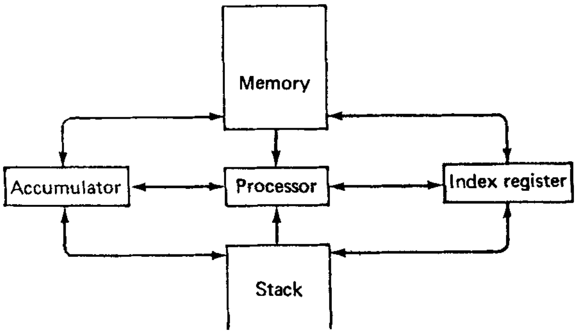


Figure 2. The architecture of the Janus family of abstract machines

The memory and the processor model the explictly named operands and the primitive operators, respectively. Details of the structure and capabilities of these components are omitted from the definition of Janus because they depend upon the particular set of primitive modes and operators provided by the abstract machine. We maintain a separate document[19] which contains complete specifications of all primitive modes and operators used in abstract machines of the Janus family. This document provides a 'library' for abstract machine designers, but does not prevent them from adding new modes or operators when necessary.

An expression is used to avoid explicitly naming the intermediate results of a computation, and hence this formation rule introduces *anonymous operands*. The accumulator and the stack of Figure 2 model anonymous operands in the same way that the memory models explicitly named operands.

Figure 2 favours target computers with a single arithmetic register, or with multiple arithmetic registers and register/storage arithmetic. Three other possible organizations come to mind:

(1) Register file (CDC 6000, 7000).

(2) Stack (Burroughs 5000, 6000).

(3) No programmable registers (IBM 1400, 1620).

(The register file machine is similar to the multiple register machine, except that it has no register/storage arithmetic. All operands must be loaded into registers before any arithmetic can take place.)

For machines of types (1) and (2), it is necessary to expand certain symbolic Janus instructions into sequences of machine instructions as shown in Figure 3. This expansion is easy to do, because the code generator need not consider the context in which the instruction occurs. Suppose, however, that Janus were essentially postfix [Figure 3(c)]. Translation for a single register machine would require that a sequence of instructions be *merged*. Contextual information would have to be kept by the code generator in order to determine that the second LOAD should generate nothing, and that its operand should be saved and attached to the ADD. Although such context-dependence presents no theoretical difficulties, it does complicate the code generator.

```
A = B+C
```

(a) A typical expression

```
LOAD    B
ADD     C
STORE   A
```

(b) Sequence of operator/operand pairs equivalent to (a)

```
LOAD    B
LOAD    C
ADD
STORE   A
```

(c) Expansion of (b) for a stack machine

Figure 3. Use of operator/operand pairs

Machines with no programmable registers have no provision for anonymous operands, and hence all operands must be given names by the code generator. Correct code can always be produced by simulating a machine with a single arithmetic register, although a slight optimization is sometimes possible by considering context. The trend in computer architecture seems to be away from machines of this type.

An array or structure is an aggregate which is recognized as a distinct entity, but whose components may be used individually as operands. Access to a component is provided by an *index*, which defines a location within the aggregate. An index might be specified by any expression, and hence may involve anonymous results. This is the reason for the Janus index register and its association with the stack: It is used in the computation of an index expression just as the accumulator is used in the computation of other expressions. The index register is distinct from the accumulator because its content is used to form operands for operations on the accumulator.

Again, the organization of Figure 2 favours a certain class of target computers—those with explicit index registers. Although this class seems to represent the mainstream of computer architecture, we have encountered three other mechanisms for component addressing:

(1) Index modification on a stack (Burroughs 5000, 6000).
(2) Indirect addressing (IBM 1620, Siemens 305, 306).
(3) Program modification (IBM 1620).

The model which we have chosen provides enough information to generate correct code in each case without retaining contextual information. (A model more closely related to any of these three mechanisms would require contextual information if code were to be generated for a machine with an index register.)

Figure 2 thus reflects the use of expressions to form complex operations from primitive operators, and the use of arrays and structures to create complex data items from primitive modes. Since there appears to be widespread agreement about how these particular formation rules should be implemented in hardware, Figure 2 also reflects the organization of contemporary computers.

There is less agreement regarding the hardware realization of conditionals, iterations and procedures. Our study indicates that these formation rules do not have a strong influence on the over-all *organization* of a machine; they are reflected in the set of *operators* which is provided. Let us consider each in turn, examining their implementation on contemporary computers, and then presenting the operators which we recommend to users of Janus.

Conditionals

Conditionals are usually provided in one or more of three ways:

(1) A conditional transfer based upon the relationship between the content of a register and zero.
(2) A conditional transfer based upon the relationship between the contents of two registers.
(3) A conditional transfer based upon a previously set condition code.

The critical questions here are whether the abstract machine should have an explicit condition code, and whether a comparison should destroy the contents of the accumulator.

If the condition code is explicit, and is not changed by every instruction, then it may be tested long after it was set. We are not aware of any useful constructs in high level languages which would generate such code. Further, we believe that the overhead associated with maintaining the contents of the accumulator and condition code on machines using method (1) is too large. There are, however, situations in which we will have unnecessary overhead if the contents of the accumulator is always destroyed by a comparison. (Such code would usually be generated by certain kinds of *case* statements.[20, 21]) Hence we advocate a middle course: provide both destructive and non-destructive comparison operators, and assume that these set a condition code which is destroyed by any instruction other than a conditional transfer. Also assume that the accumulator is destroyed by any transfer of control except one following a non-destructive comparison. The target of such a transfer can be defined by a special pseudo, thus permitting the implementor to save the accumulator at the non-destructive comparison operator and restore it after the transfer. We will illustrate this approach in more detail later.

Iterations

An iteration may proceed under count control, or it may continue until a certain condition holds. The former is a special case of the latter, but it is sufficiently important to merit separate consideration. Some computers have special instructions (e.g. IJP,[22] BXLE, BXH[23]) which will modify a register, test it and transfer control if some condition is met. Other computers have special memory locations which perform similar functions.[24] Moreover, some languages[21, 25] provide iterations under count control in which the controlled variable may be anonymous. Even if the controlled variable is not anonymous, the programmer is not permitted to make assignments to it.

This indicates that the abstract machine model might provide an iteration counter which can be incremented and tested by a transfer instruction. We have not incorporated such a register because we have not been able to formulate the instructions to manipulate it in such a way that they apply to most computers.

Procedures

A procedure invocation involves both parameter passing and status saving. There are four common hardware mechanisms for status saving:

(1) Relevant status is placed on a stack by the hardware when a subroutine jump is executed (Burroughs 5500, ICL KDF 9).
(2) Relevant status is placed in a register by the hardware when a subroutine jump is executed (Data General NOVA, UNIVAC 1108, IBM 360).
(3) Relevant status is placed in memory by the hardware when a subroutine jump is executed. The memory location bears some fixed relationship to the target of the subroutine jump (CDC 3000, 6000, XDS 940, UNIVAC 1108).
(4) A separate instruction is provided for saving the relevant status (GE 645).

The makeup of the ‘relevant status’ depends entirely upon the computer. At the least, it is the return address.

Because of the diversity in status saving methods, one must use a high level model for a procedure call. We advocate three instructions:

(1) *Return jump* is used in the calling program to actually invoke the procedure.
(2) *Link* appears as the first executable instruction of the procedure body.
(3) *Return* is used in the procedure body to actually return from the procedure.

These instructions are interdependent, a fact which may be used by the implementor to match any one of the four status saving methods mentioned above.

In some cases the procedure call mechanism provided by the target machine is such that there is no difference in cost between recursive and non-recursive calls. Unfortunately, this is not true in general. Most modular programs employ one or more short procedures which are never used recursively. Calls to these procedures often appear in inner loops, where the overhead required for recursion builds rapidly. If a program requires recursive procedures, we strongly suggest *two* versions of return jump, link and return—recursive and non-recursive.

Parameter mechanisms (reference, value, etc.) are primarily determined by the source language. We will not discuss them in detail except to say that Janus operators must be able to manipulate parameters as addresses or values. The location of parameters in the calling sequence is system-dependent; our study has shown that there are three common techniques:

(1) Parameters are stored in the calling program in the vicinity of the return jump.
(2) Parameters are stored in a fixed area of the called program.
(3) Parameters are stored in registers or on a stack.

A diversity of parameter mechanisms, like the diversity of status saving methods, demands a high-level model. We use two special pseudos:

(1) CALL marks the beginning of the code required to compute arguments at run time. (For example, to obtain the address of A(I) when arguments are passed by reference.)

(2) CEND marks the end of a set of storage reservation pseudos which define the argument list.

These pseudos, in conjunction with the three procedure call instructions discussed above, can be used to create procedure calls conforming to a wide variety of conventions. We shall present a detailed example and discuss various realizations later.

Other formation rules

As far as we know, there are no common hardware mechanisms for providing coroutine linkage. Thus the designer of an abstract machine is free to provide coroutine linkage in any way he sees fit. We would suggest that an explicit coroutine call operation be provided, but we make no attempt to specify its properties. When (and if) hardware coroutine linkage is developed, we shall reconsider the question.

Reference modes are represented by addresses or by descriptors.[26,27] A primitive mode ADDR may be used to describe either representation. A 'load immediate' instruction creates an entity of ADDR mode which references the operand of the instruction, and leaves it in the accumulator. A pseudo which allows one to preset a reference in memory is also required.

The three remaining formation rules for mode creation listed in Table I have only a minor effect on the abstract machine. They are primarily the concern of the compiler, which uses domain restriction to provide additional information for generating checking code during debug runs and models the enumerated elements of a new mode with small integers. Domain restriction could also result in new modes for the abstract machine model if the designer felt that significant storage economies would be possible.[28]

THE SYMBOLIC REPRESENTATION OF JANUS

The symbolic Janus program is an encoding of the information which passes across the interface of Figure 1(a). Our design goal was to present the information in a form which would simplify the process of producing assembly language for the target computer. In this section we shall discuss the form of the encoding, explaining the need for each piece of information. The next section gives examples of symbolic Janus programs. Although it would certainly be *possible* for a human programmer to write Janus, we do not advocate such an approach. We assume that the code will always be produced as the output of some translator.

Janus has two basic formats for executable instructions:

operator model mode2
operator mode reference

The first of these is used for instructions which may be broadly classed as mode conversion (fix, float) and value conversion (negate, truncate) operations which modify the contents of the accumulator. Model and mode2 are the initial and final modes, respectively, of the accumulator contents. The second instruction format is used when an operand other than the accumulator must be specified. We distinguish three major classes of references:

references to explicitly named operands, references to anonymous operands and references to constant operands. Each class presents unique problems, which we shall discuss in the following subsections.

Explicitly named operands

Each reference to an explicitly named operand specifies a symbol and an index. (If the operand is not a component of an aggregate, then the index is null.) Normally, the attributes of each symbol would be specified by declaration and stored in a dictionary by the translator. The symbol would serve as a key to the dictionary, allowing the translator to access the attributes of the operand each time it was used. However, dictionary lookup is a time and space consuming process for STAGE2 which we wished to avoid if possible.

Examination of the usual attributes of an operand revealed two which influenced the translation of the reference into machine code:

(1) Category.

(2) Address.

The category provides an interpretation of the operand—it tells what sort of operand is being represented. For example, categories can be used to distinguish arguments, formal parameters and particular storage areas (local, global, dynamic, etc.). Often these operands will require different translations, or significant optimization may be possible for some on certain machines.[28] Such special treatment can be provided only if it is possible to distinguish the operands which require it. Hence we decided to include the category explicitly in each reference.

On most machines the value of the address does not affect the sequence of instructions needed to accomplish the reference. Provided that the symbol in the Janus instruction is acceptable to the target machine's assembler, it may be used unchanged in the output code. Under these circumstances no dictionary lookup is required to establish the address during the STAGE2 run. If either condition is violated, then a lookup will be necessary.

It is useful to separate the index of an aggregate reference into two parts: the *fixed offset* (whose value may be obtained from the program text) and the *variable offset* (whose value must be determined during execution). On many computers the base address and fixed offset may be combined at translation time to form an 'effective base address'. The final address is then obtained at run time by addition of the variable offset. Such a procedure will not work on a machine which references arrays via descriptors.[26, 27] There, code must be generated to combine the fixed and variable offsets to form the index at run time. The final address is then obtained from the contents of the descriptor and this index.

We have therefore given instructions which reference explicitly named operands the following general form:

operator mode category symbol(fixed)variable

Either or both of the offset specifications may be omitted.

Anonymous operands

A reference to an anonymous operand is a reference to the top element of the stack. If the target computer does not have a hardware stack, then the Janus stack must be simulated in memory. This means that anonymous operands require much the same information as explicitly named operands. (If the target computer does have a hardware stack, the extra information is simply ignored.) All references to anonymous operands are in the same category, and hence the category field of the instruction can be used to indicate that the operand is anonymous. The address specifies a location within the current frame of the simulated stack; component references are impossible and thus no index is required.

The only problem is that entities of different modes require different amounts of storage and, in some cases, must be aligned on different address boundaries. It would be wasteful to simulate the stack by an array of fixed size elements, and hence the actual address corresponding to a particular operand must be determined by the contents of the stack at the time the operand is pushed on to the stack. Most of the bookkeeping associated with this storage allocation can be performed by the analyser;[29] only the actual address assignment must be deferred until the characteristics of the target machine are known.

Three pieces of information are sufficient to determine the address at which to store a new element:

(1) The address of the previous element.
(2) The size of the previous element.
(3) The alignment of the new element.

Items (1) and (2) can be combined to determine the first address beyond the previous element, and item (3) can be used to determine the proper boundary. The address of the new element is then assigned to the symbol for the anonymous operand, which would be used to provide item (1) for the next new element. (Note that both the size and alignment of an item can be determined from its mode.)

We have therefore given instructions which reference anonymous operands the following general form:

operator mode STACK symbol(size)previous

Both 'size' and 'previous' would be omitted if the operand were the first on the stack.

Constant operands

A reference to a constant operand may or may not involve a memory reference on the target machine: the value can often be incorporated into the instruction itself if the constant satisfies certain machine-dependent constraints. When this cannot be done, the constant must be placed in memory. All references to constant operands are in the same category, and hence the category field of the instruction can be used to indicate that the operand is a constant. It is also useful to associate a symbol with the constant in case the target machine's assembler is incapable of handling literals.

We distinguish four types of constants:

(1) As-is.
(2) Expression.
(3) Symbolic.
(4) Character code.

An as-is constant is independent of the target computer, and its value can be placed directly into the Janus code. All others are machine-dependent, and hence must be expressed symbolically.

A constant of type (2) is usually associated with the addressing structure of the target computer. Its complexity is limited by the translator; STAGE2 permits integers, variables with integer values and the operators +, −, *, and /. Parentheses may be used and nested to any depth (subject to storage limitations). Each Janus mode identifier is assumed to be a variable whose value is the number of target machine address units occupied by an entity of the corresponding mode.

Some algorithms use machine-dependent constants which are unrelated to the addressing structure. The range reduction used in computing EXP(X), for example, requires the natural logarithm of the base of the computer's arithmetic.[30] Such constants may not satisfy the constraints on variables allowed in expressions, and hence must be treated differently. Their values are preset in the memory of the translator by a pseudo, and are substituted into the Janus instruction when a type (3) reference is detected.

Character codes could be handled using a type (3) reference, but we have found that this requires excessive amounts of translator memory. It is a simple matter for the translator to compute the integer equivalent of a character,[5] and hence we treat such constants separately.

We have therefore given instructions which reference constant operands the following form:

operator mode CONST symbol() type value

Storage reservation

To reserve storage for an operand we must know the mode of the operand, whether it is an array and the number of array elements. (Simple variables are not equivalent to arrays of length 1 on a computer which references arrays via descriptors.[26, 27]) If the contents of the reserved area are to be initialized, most assemblers require that the initial values be presented at the time the storage reservation is made. This requirement presents a problem only when the operand is an array whose elements are to have different initial values. In all other cases, a constant (of the form discussed in the last subsection) can be attached to the reservation pseudo.

The storage requirements of the entire array should be stated in a single pseudo so that a descriptor for the array can be constructed. If all elements have the same initial value, that value can be attached to the pseudo. Otherwise, the pseudo is flagged to indicate that initialization follows. The initial values can then be defined by a sequence of pseudos, each of which sets or skips a block of elements. The total number of elements specified by the sequence should equal the number of elements in the array.

We have therefore given the storage reservation pseudo the following general form:

SPACE mode category symbol(elements)flag type value

If 'flag' is present, initial value specifications follow this pseudo; 'symbol' and 'flag' will both be omitted on those specifications.

EXAMPLES OF JANUS CODE

Our purpose here is to make the discussion of the previous sections concrete, and to indicate the correspondence between Janus and machine code. We shall not attempt to illustrate all of the features of the language, nor shall we dwell upon the details of representation.

The SPACE pseudo

Figure 4 contains several storage reservations, both with and without initialization. The constant types are flagged by 'A' (as-is), 'E' (expression), 'M' (symbolic) and 'C' (character code). A plus is used as the 'initialization to follow' flag. We have found that generated symbols of the form 'Gn' are accepted by most assemblers, and hence we can avoid a dictionary lookup in STAGE2.

```
SPACE INT LOCAL G1( ).               SIMPLE VARIABLE

SPACE INT LOCAL G2( 3).              THREE–ELEMENT ARRAY
```

(a) Reservation without initialization

```
SPACE INT LOCAL G3( ) C X.           CHARACTER CODE

SPACE REAL LOCAL G4( ) M LNBASE.     SYMBOLIC CONSTANT

SPACE INT LOCAL G5(15) A 6.          FIFTEEN IDENTICAL ELEMENTS
```

(b) Reservation with initialization

```
SPACE INT LOCAL G6(4)+.              DECLARE ARRAY, INITIALIZATION FOLLOWS

SPACE INT LOCAL (1) A 0.             FIRST ELEMENT IS INITIALIZED

SPACE INT LOCAL (1) .                SECOND ELEMENT IS NOT

SPACE INT LOCAL (2) A 1.             THIRD AND FOURTH ARE
```

(c) Separate initialization of array elements

Figure 4. Use of the SPACE pseudo

A Janus procedure

The procedure of Figure 5 calculates the square root of a positive real number, using Newton's iteration.[30] We have simplified the algorithm somewhat in order to concentrate on the features of Janus.

The first line specifies the mode of the result and the number of parameters, as well as the name of the routine. In most cases this information will not be used for the translation of BEGIN; it is included to ease the implementation of certain linkage conventions.

A SPACE declaration is given for each parameter. These declarations may or may not reserve storage. They serve to identify the mode of the parameter and to associate a symbol with the parameter position. Declarations of parameters are distinguished by the category PARAM, and hence may be treated specially by the translator.

LINKN is a special form of LINK, which conveys the additional information that this procedure does not invoke other procedures. Some optimization may be possible in this case if parameter addresses are passed in registers. A single STAGE2 macro can be provided to translate both LINK and LINKN, thus ignoring the added information. This would normally be done for the first implementation in order to bring a program up quickly.[31]

CMPNF is another example of the use of additional information attached to an operator. A non-destructive comparison, CMPN, is used to check the content of the accumulator without destroying it. On a multiple-register computer, the Janus accumulator might be assigned to different registers in different parts of the program. If the value of the accumulator is used after a jump, and if we are translating the Janus code in a single pass, then we must insure that the accumulator occupies a standard register before the jump. The F indicates, however, that the value of the accumulator is used only in the 'fall-through' path, and hence its position need not be standardized at this point. The LOC pseudo defines a label and also indicates whether the contents of the accumulator and index register are significant at that point.

```
BEGIN REAL PROC SQRT(1)          SQRT RETURNS A REAL AND HAS ONE PARAMETER
SPACE REAL PARAM G92( ).         DECLARE THE FORMAL PARAMETER
SPACE REAL LOCAL G93( ).         DECLARE A LOCAL VARIABLE
LINKN REAL PROC SQRT(1).         PERFORM LINKAGE DUTIES IF NECESSARY
LOAD REAL PARAM G92( ).          ACCESS THE VALUE OF THE FORMAL PARAMETER
CMPNF REAL CONST G22( ) A OEO.   DOES NOT DESTROY THE ACCUMULATOR CONTENTS
JLT, I INSTR CODE G99( ).        ABORT THE RUN ON A NEGATIVE ARGUMENT
LOC REAL VOID G93( ).            ACCUMULATOR CONTENTS REAL, INDEX IRRELEVANT
STORE REAL LOCAL G93( ).         SAVE THE CURRENT GUESS
LOAD REAL PARAM G92( ).          RECALL THE VALUE OF THE FORMAL PARAMETER
DIV REAL LOCAL G93( ).           DIVIDE BY THE CURRENT GUESS AT THE ROOT
ADD REAL LOCAL G93( ).           AVERAGE THE RESULT WITH THE CURRENT GUESS
DIV REAL CONST G88( ) A 2EO.     TO GET A NEW GUESS
CMPN REAL LOCAL G93( ).          DOES NOT DESTROY ACCUMULATOR CONTENTS
JNE, I INSTR CODE G98( ).        REFINE THE GUESS AGAIN IF NECESSARY
RETURN REAL PROC SQRT(1).        ELSE RETURN WITH RESULT IN ACCUMULATOR
LOC VOID VOID G99..              ACCUMULATOR AND INDEX CONTENTS IRRELEVANT
MSG STRNG CONST G100( ) A NEGATIVE ARGUMENT FOR SQRT.
ABOUT REAL PROC SQRT(1).         ABANDON THE EVALUATION OF THE PROCEDURE
END SQRT.
```

Figure 5. A Janus procedure

The operand of the conditional jump is the specified address, not the contents of that address, and hence an immediate modifier (indicated by ',I') is used.

Procedure call

One of the problems with procedure calls is that of insuring *modularity*:[32] in order to run programs written in Janus with those written in other languages one must be able to translate a Janus procedure call into the standard calling sequence assumed by the other languages. Thus it is extremely important to be able to *recognize* parameter setup and parameter use in the Janus code. If these constructs can be recognized, then translation rules can be written to match virtually any conventions.

As a concrete example, consider the procedure call of Figure 6(a). The first and third arguments are to be passed by value, while the second and fourth are to be passed by reference. Computation is required to obtain the third and fourth arguments. We assume that the procedure returns a value in the accumulator.

Figure 6(b) shows the Janus version of the call. (We have used the variable names instead of generated symbols for clarity.) Two specifications of the arguments are given. The first, lying between CALL and RJMP, shows how they are computed. The second, lying between RJMP and CEND, is a list of argument addresses.

The translation of D[I + 3] reflects the fact that an array index is an integer which must be multiplied by the number of address units per element before being used. Multiplication of the fixed offset can be carried out at translate time; a separate Janus instruction performs the multiplication of the variable offset. (The '+' in the variable offset field of the reference specifies an anonymous operand in the index register.)

```
F(1,A,B+C,D[I+3])
```

(a) A procedure call

```
CALL REAL PROC F( )
ARGIS INT CONST C1( ) A 1
ARGIS, I REAL LOCAL A( )
LOAD REAL LOCAL B( )
ADD REAL LOCAL C( )
STARG REAL TEMP T1( )
LDX INT LOCAL I( )
MPX INT CONST C2( ) E REAL
LOAD, I REAL LOCAL D(3* REAL)+
STARG ADDR ARG L1(3* ADDR)
RJMP REAL PROC F( )
SPACE ADDR ARG L1(4)+
SPACE ADDR ARG (1) A  C1
SPACE ADDR ARG (1)  A  A
SPACE ADDR ARG (1)  A  T1
SPACE ADDR ARG (1)
CEND REAL PROC  F( )
```

(b) Janus code for (a)

Figure 6. Example of a procedure call

We now indicate how this procedure call would be translated for several interface conventions:

Stack

Hardware. CALL produces a 'mark stack' instruction and ARGIS places a value on the stack. (The second line of Figure 6(b) places the integer constant 1 on the stack, while the third line stacks the address of the local variable A.) STARG generates nothing, since the value of B + C and the address of D[I + 3] will have already been placed on the stack. RJMP generates the jump to the procedure and then causes the translator to ignore succeeding statements up to and including CEND.

Software. Same as hardware example, except that STARG must generate code to transfer a value from the register in which it is computed to the appropriate location on the simulated stack. The operand of the STARG operation is ignored.

List of addresses following the jump to the procedure

CALL and ARGIS generate no executable code, although an ARGIS with a constant operand would cause the translator to subsequently output a declaration for the constant. STARG produces a normal store. RJMP becomes the subroutine jump and the SPACE declarations result in an initialized argument list. CEND either generates nothing or plants the target label for a jump over the argument list.

List of addresses within the procedure body

CALL sets up the location of the argument list. ARGIS and STARG generate code to move information into it. For example, the STARG in line 6 would cause the value of $B+C$ to be stored in T1 and the address of T1 to be placed in the third position of the argument list. RJMP generates the return jump to the subroutine and causes the translator to ignore subsequent lines up to and including CEND.

List of values within the procedure body

Same as the previous list, except that ARGIS and STARG store values instead of addresses into the argument list.

The calling sequence will not easily handle the case in which values are stored following the jump to the procedure. However, we suggest that this form is unlikely to be used in practice: access to the arguments from within the procedure body would involve some kind of indexing or indirection, and space to store the arguments would be needed at every call.

REVIEW OF RELATED WORK

This section contains short descriptions of three projects whose goals are related to those of Janus. We have presented them in a similar format, related to our discussion of Janus, in order to make comparison possible. Each description concludes with a brief critique.

UNCOL

UNCOL was originally proposed[13-15] to solve the $n \times m$ *translator problem*: a straightforward implementation of n languages on m machines requires one translator for each language/machine pair. This number could be reduced to $n+m$ if a *un*iversal *c*omputer-*o*riented *l*anguage were available: One analyser would be required to translate each of the n languages into UNCOL, and one code generator would be required to translate UNCOL for each target computer. Since the analysers themselves were to be written in UNCOL, the entire system could be moved to a new computer simply by writing a code generator for that machine.

Steel[16] is the only author who gives any concrete proposals for the implementation of UNCOL, and his specification is far from complete. We surmise that the organization of the abstract machine was similar to ours, with a ' . . . generalized and exceedingly pliable accumulator-like gadget . . . ' but no stack. Each instruction consisted of an operator/operand pair. Steel listed approximately twenty operators, and pointed out that the meaning of each depends upon the mode of its operand. (He also provided modifiers for each operator which specified that the operand should be negated or its absolute value taken.)

The operand of an UNCOL instruction was specified by location only. No mode or category information was given in the instruction; a separate 'data description' was used. (Steel mentioned the data description only briefly, and did not present any details.) A location consisted of a base and an index, and could be the start of a chain of indirect references. It was possible to specify a fixed depth of indirection, or to allow indirection to be controlled by the contents of the referenced locations. The base and index were themselves locations, and hence could also be subject to indexing and indirection.

A pair of brackets, DEFINE and END DEFINITION, were to be used in an unspecified manner to identify groups of instructions which implemented certain complex operations. These brackets were to be recognized by the code generator if the target computer had hardware facilities to perform the complex operation. Their function was similar to that of a macro definition, except that the complete expansion was present in the UNCOL text at each use.

Finally, Steel mentioned that additional declaratives were being considered. These declaratives would specify flow information which was deduced from the original program, and would permit the code generator to optimize the object program. No declaratives were given in the paper; Steel said that work was then under way to determine a suitable set.

The UNCOL project was abandoned before a more complete specification of the language was published, but we doubt that the designers rejected the basic idea. They may have found that compilation techniques were not clearly understood, and that adaptable translators were difficult to write. We feel that a decade of advances in these areas, especially the latter, enhance the chances for Janus to succeed. Powerful macro processors such as STAGE2 allow us to modify translation techniques as we experiment with Janus features.

SLANG

SLANG[17] was the result of a project initiated early in 1960 for the purpose of developing a programming language suited to the task of writing compilers. The approach was similar to that of UNCOL, except that the analysers had access to some 'general information' about the target machine. This included such items as the characteristics of storage (word size, addressability), of instructions (number per word, can they overlap word boundaries?) and of registers (how many, what capabilities?). Using this information, the analyser produced a sequence of instructions for an *E-machine*. Sibley claimed that this output could not be considered an UNCOL because the analyser would produce different instruction sequences when give different machine descriptions.

The organization of the E-machine was not specified; we surmise that it had only a memory and a processor. Each instruction consisted of an operator and an appropriate number of operands. Sibley listed fifty-five operators, and did not say that the meaning of an operator depended upon the modes of its operands. Since he had distinct operators for index arithmetic, we assume that a particular operator could accept operands only of a particular mode.

Each operand of an E-machine instruction was specified by a single symbol or integer. No mode information was given because it was irrelevant for the remainder of the translation. (The effect of the operand mode had already been taken into account by the analyser.) Since the analyser had access to the details of the addressing structure on the target machine, it could generate the instructions necessary to manage index registers; hence there was no need to specify indexing or indirection for the operand itself.

The set of E-machine instructions was not fixed. If a particular target computer had hardware facilities to perform some complex operation, a new instruction would be provided.

This instruction would be produced by the analyser only for machines on which it was appropriate. Sibley did not give details about how the analyser could make this decision.

Finally, Sibley mentioned that additional pseudos could be used for conditionally including sequences of instructions in the generated code. These pseudos were apparently not used to provide flow information, since all global optimization was done by the analyser. 'Peephole optimization'[33] was carried out as the E-machine instructions were translated, and conditionals might have been useful there.

SLANG probably failed because the problem of an adaptable analyser was never adequately solved. We believe that this approach is a dead end, and that the adaptability must come in the translation from the intermediate language to machine code.

OCODE

OCODE[34] is the intermediate language used for the BCPL[35, 36] compiler. The design criteria were similar to those of Janus, except that there was no need for extensibility. Thus OCODE is a single language rather than a family, and the characteristics of the abstract machine are completely specified.

The OCODE machine consists of a memory, a processor and two memory address registers, S and P. The memory is an array of integer elements, represented by bit patterns. Each instruction consists of an operator and an appropriate number of operands. Richards lists forty-eight operators, none of which is subject to any kind of modification.

Each operand of an OCODE instruction is specified by a single symbol or integer. No mode information is needed because only one mode is available. Many of the operators do not specify operands; they access their operands relative to the location addressed by S. Thus memory is used as a stack, with S addressing the current top. P addresses the base of the current *stack frame* (the area of memory associated with a procedure invocation), and other operators have non-negative integer operands which are interpreted either as offsets from this point or as absolute memory locations.

Richards mentions that one of the best ways for the OCODE translator to perform local optimization is to simulate the state of S and emit instructions 'only when it becomes necessary to simplify the simulation'. He also states that one advantage of this technique is that it is 'relatively machine independent and hence much of [an OCODE translator] for one machine can be used . . . for another'.

As of October 1970, OCODE had been used to transport the BCPL compiler to between ten and twenty different machines. Richards does not list them, but we suspect that none had a hardware stack. Such computers usually do not provide mechanisms for altering the stack pointer and base of the current stack frame explicitly, but both operations are available in OCODE. Nevertheless, we feel that OCODE has been successful in its intended application and has demonstrated the feasibility of the intermediate language approach to translation.

CONCLUSION

Janus has now been implemented, via the STAGE2 macro processor, on two computers. We estimate that approximately two man-weeks of effort are required to construct the macros for a new machine. This would be the effort required to implement the first piece of Janus software; subsequent software would require only the additions to the original set of macros.

To verify the viability of our approach, we decided to modify the code generation routines of the Pascal and BCPL compilers. When these modifications are complete, we will be able to translate programs written in Pascal and BCPL into Janus. Thus we will be able to compare machine code generated directly from these languages with code generated via Janus. Since the compiler's analysis phase will be the same in both cases, any differences can be attributed to the use of the intermediate code. (Pascal and BCPL were chosen because their compilers were readily available and easily modified.) Preliminary results from the Pascal project[37] indicate that code produced by using Janus as an intermediate language is approximately the same size as that produced by the standard compiler for the CDC 6400,[28] and runs about 10 per cent slower.

ACKNOWLEDGEMENT

This work was supported by the National Science Foundation under Grant GJ-32471.

REFERENCES

1. R. J. Orgass and W. M. Waite, 'A base for a mobile programming system', *Comm. ACM*, **12**, 507 (1969).
2. W. M. Waite, 'Building a mobile programming system', *Computer J.* **13**, 28 (1970).
3. P. C. Poole and W. M. Waite, 'Machine independent software', *Proc. ACM Second Symposium on Operating System Principles*, 19 (1969).
4. W. M. Waite, 'The mobile programming system: STAGE2', *Comm. ACM*, **13**, 415 (1970).
5. W. M. Waite, *Implementing Software for Non-numeric Applications*, Prentice-Hall, Englewood Cliffs, N.J., 1973.
6. M. C. Newey, P. C. Poole and W. M. Waite, 'Abstract machine modelling to produce portable software—a review and evaluation', *Software—Practice and Experience*, **2**, 107–136 (1972).
7. P. J. Brown, 'The ML/1 macro processor', *Comm. ACM*, **10**, 618 (1967).
8. C. Strachey, 'A general purpose macrogenerator', *Computer J.* **8**, 225 (1965).
9. P. J. Brown, 'Using a macro processor to aid software implementation', *Computer J.* **12**, 327 (1969).
10. R. E. Griswold, *The Macro Implementation of SNOBOL4*, W. H. Freeman, San Francisco, 1972.
11. M. I. Halpern, 'Machine independence: its technology and economics', *Comm. ACM*, **8**, 782 (1965).
12. P. C. Capon, D. Morris, J. S. Rohl and I. R. Wilson, 'The MU5 compiler target language and autocode', *Computer J.* **15**, 109 (1972).
13. O. Mock, J. Olsztyn, J. Strong, T. Steel, A. Tritter and J. Wegstein, 'The problem of programming communications with changing machines: a proposed solution', *Comm. ACM*, **1**, 12 (1958); **1**, 9 (1958).
14. T. B. Steel, Jr., 'UNCOL, universal computer oriented language revisited', *Datamation*, **6**, 18 (1960).
15. T. B. Steel, Jr., '*UNCOL*: the myth and the fact', in *Annual Review in Automatic Programming*, Vol. 2 (Ed. R. Goodman), Pergamon Press, New York, 1961, p. 325.
16. T. B. Steel, Jr., 'A first version of UNCOL', *Proc. AFIPS WJCC*, **19**, 371 (1961).
17. R. A. Sibley, 'The SLANG system', *Comm. ACM*, **4**, 75 (1961).
18. M. V. Wilkes, 'The outer and inner syntax of a programming language', *Computer J.* **11**, 260 (1968).
19. *Janus Summary*, Dept. of Electrical Engr., Univ. of Colorado, Boulder, Colorado, 1973.
20. W. A. Wulf, D. B. Russell and A. N. Habermann, 'BLISS: a language for systems programming', *Comm. ACM*, **14**, 780 (1971).
21. A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck and C. H. A. Koster, 'Report on the algorithmic language ALGOL 68', *Numerische Mathematik*, **14**, 79 (1969).
22. *3800 Computer System Reference Manual*, 600162300, Control Data Corp., St. Paul, Minnesota, 1968.
23. *IBM System/360 Principles of Operation*, A22-6821-5, IBM Corp., Poughkeepsie, N.Y., 1967.
24. *How to Use the Nova Computers*, DG NM-5, Data General Corp., Southboro, Mass., 1971.
25. M. Rain, *MARY Formal Syntactic Specification*, Computing Centre, Technical University of Norway, Trondheim, 1972.
26. J. Iliffe, *Basic Machine Principles*, American Elsevier, New York, 1968.
27. *B5500 Information Processing Systems Reference Manual*, 1021326, Burroughs Corp., Detroit, Mich., 1967.

28. N. Wirth, 'The design of a PASCAL compiler', *Software—Practice and Experience*, **1**, 309–333 (1971).
29. M. Kanner, P. Kosinski and C. L. Robinson, 'The structure of yet another Algol compiler', *Comm. ACM*, **8**, 427 (1965).
30. J. F. Hart (Ed.), *Computer Approximations*, John Wiley, New York, 1968.
31. P. J. Brown, 'Levels of language for portable software', *Comm. ACM*, **15**, 1059 (1972).
32. J. B. Dennis, 'Modularity', in *Advanced Course on Software Engineering* (Ed. F. L. Bauer), Springer-Verlag, Berlin, 1973.
33. W. M. McKeeman, 'Peephole optimization', *Comm. ACM*, **8**, 443 (1965).
34. M. Richards, 'The portability of the BCPL compiler', *Software—Practice and Experience*, **1**, 135–146 (1971).
35. M. Richards, *The BCPL Reference Manual*, Tech. Memo. 69/1, Computer Laboratory, Cambridge, 1969.
36. M. Richards, 'BCPL: a tool for compiler writing and system programming', *Proc. AFIPS SJCC*, 557 (1969).
37. L. B. Weber, *A Machine Independent Pascal Compiler*. M.S. Thesis, Univ. of Colorado, Boulder, 1973.