Department of Computer Science

Memorandum

IMP CONVERGENCE II

To G.E. Thomas, P.D. Stephens, G.E. Millard (ERCC)
J.P. Gray, P.S. Robertson, I.A. Young (Lattice Logic)
P.D. Schofield, S. Michaelson, D.J. Rees (Computer Science)

From H. Dewar

Date 26th October, 1982.

Background

The first IMP convergence exercise, which started in 1980 and has been reported on in Alan Anderson's documents of 20/1/81 and 31/3/81, made substantial progress towards eliminating the obstacles to moving IMP programs between one implementation and another. A number of residual differences remain; these are not on the whole as major as those that were overcome in the first exercise, but they can pose significant problems for IMP program portability. The production of two new compilers for the language, in the Regional Centre for the Perq and in the Computer Science Department for the Motorola 68000, makes it timely to seek further progress.

The purpose of this document is to set out as many of the significant points of difference between the main existing compilers as I have been able to discover. In order to do so, I have drawn on Peter Robertson's latest manual covering IMP77 for Vax/VMS (hereafter referred to as Vax IMP) and John Murison's preliminary manual for IMP80 on the 2972 (hereafter Emas IMP), as well as Alan Anderson's reports. However, as there have been some developments since these documents were produced, I have used my own experience and specially conducted tests to try to produce an up-to-date picture of the current releases of these two implementations. The picture is unlikely to be totally accurate, but I hope that the number of omissions and mis-representations is small.

As well as identifying the points of difference, I have put forward specific proposals for almost every case as a basis for discussion. These are in line with the course I should like to follow in the M68000 version.

Objectives

The proposals in this document are directed to those who wish IMP well. Opinions differ on the suitability of IMP as a general-purpose programming language, and on the desirability of using it for applications at large even if it is deemed suitable. What seems to me not open to serious question is that it has been of considerable value as a language for implementing a wide range of system software, from operating systems, through compilers and basic packages, to standard utilities like editors and formatters. The advantages which have derived from this approach at Edinburgh deserve to be more firmly emphasised. It is still typically the case for systems in general that basic software is written either in Assembler or in a mixture of various languages; in either case, the development and support process is much more onerous than is the case when a single high-level implementation language is employed. The Emas papers present some of the considerable advantages.

There can be an additional bonus. It can prove possible to move some of these software components (those which are not inevitably too system dependent) from the system for which they were developed to another one. When users (quite reasonably) comment on the non-portability of IMP programs to other sites, because the language is not in widespread use, they might also reflect that it has in fact enabled a number of pieces of software, not normally regarded as portable, to be transferred between different systems in Edinburgh.

One of the most important factors in opening up this possibility is that the system implementation language, which must by definition permit some of the normal constraints of high-level languages to be broken so that access may be permitted to machine and system features, should nonetheless offer ways of doing so which limit and localise absolute machine or system dependence. For example, the availability of typed pointer variables is an aspect in which IMP is superior to, say, BCPL. Our understanding of how to specify operations in a well-defined fashion (as opposed to a system-defined fashion), and of what facilities can be implemented efficiently, has increased with experience, so that there is now less reason to tolerate in the language features which are logically insupportable. If we are prepared to make it a high enough priority, we should be able to achieve even greater portability of system software in the future, and at the same time improve IMP's qualities as a general-purpose language.

Of course, the objective of greater portability is not a matter of language (or programming discipline) alone. It also depends critically on the availability from the underlying operating systems of a broadly comparable set of capabilities (and the absence of certain crucial restrictions). I believe that it would be a most worthwhile enterprise at this stage to try to define a standard system interface which could be supported across all the main machines at Edinburgh. This, however, goes beyond the scope of the present document.

The IMP language

There is room for only one IMP language. This is not to say that a single specification should be frozen for all time, since it is one of the compensating virtues of a 'local' language that it can be refined and evolved. Nor is it to say that there can never be compelling reasons for the existence of differences between implementations. What it does mean is that there must be an independent definition of the language itself, not a new language manual for each implementation.

Proposal

The production of a proper IMP language manual should be given high priority. It should be a joint project between (at least) ERCC and EUCSD; it should be aimed at a somewhat higher conceptual level than any of the existing manuals; and there should be a prohibition on any mention in the body of the text of specific implementation restrictions and differences. It would be desirable to aim for publication by EUP.

SPECIFIC PROBLEM AREAS

Predicates

Vax IMP extends the class of procedures to include predicates as well as routines, functions and maps; Emas IMP does not. There is no doubt that for certain kinds of program which make extensive use of procedure calls for carrying out tests, the use of predicates makes the conditional clauses much neater and more readable. Of course, the same effect can be achieved by using functions and testing the value of the call to be zero or non-zero, but this is clumsier and less clear.

Proposal

Either predicates should be included in the language as such or the definition of condition should be modified to the effect that a single arithmetic expression by itself should be permitted as a condition, with the effect of a test non-zero. The latter does not involve an extra procedure type and is a more general facility, at the expense of some loss of error-detecting redundancy.

Integer ranges

Current compilers directly reflect the specific capabilities of their target processor in their choice of integer ranges. For 16-bit operands, Vax IMP provides the signed range of -32768 to 32767 (short); Emas IMP provides the unsigned range of 0 to 65535 (half). This is proving to be one of the major sources of difficulty in transferring programs, not least in cases where the values involved are always within the common subrange 0 to 32767 or where, for other reasons, the distinction is unimportant. Such difficulties will multiply with the introduction of microprocessors like the Motorola 68000 which directly support signed 8-bit values. Tad Pinkerton long ago urged the desirability of permitting the programmer to specify in a declaration the precise range of values that a variable could assume. Not only is this superior in terms of program specification, but it leaves it up to the compiler to determine the appropriate storage format. There is a known difficulty in this approach, arising from the fact that all current compilers are obliged to treat range differences as type differences for the matching of name parameters, but it might be no bad thing to put some pressure on the removal of that restriction. Even in a partially-supported form, it seems to be the only way forward.

Proposals

(a) It should be permitted to specify an explicit range within an integer declaration in the form:

integer (LOWER:UPPER) .....

(b) Man-sized compilers should be expected to support as storage formats both signed and unsigned 16-bit values and both signed and unsigned 8-bit values, albeit less efficiently for the cases not covered by hardware;

(c) The keywords short, half and byte should continue to be available as shorthand for familiar ranges, along, perhaps, with mite (for signed 8-bit range) and bit.

Records

Records got off to a bad start in IMP, with an unrortunate choice of syntax for their declaration. They are still in some ways second-class types (in varying ways in the two compilers), and this should be remedied in what are, on the whole, obvious ways.
A significant deficiency at present is the absence of a facility for initialising own and constant records and of a facility to construct a record from individual components. The main problem is in devising a satisfactory syntax, especially taking account of the complexity introduced by alternative formats. Peter Roberston has suggested a form in which the individual component name is specified with each value; this has the advantage of precision and flexibility, but may be regarded as excessively cumbersome for simple cases. It could be that an unadorned list of values, with enforced use of first alternatives, would meet the need.

Proposals

(a) Some method of achieving partial record assignment (absent from Vax IMP) should be included (see discussion of Jam Assignment below);
(b) Record maps and functions, record format specs, and the star specifier for record names and maps (absent from Emas IMP) should be included;
(c) Initialisation of own and constant records should be permitted in a form to be agreed;
(d) The name of a record format should be usable as a constructor function, the exact syntax to be agreed;
(e) The convenient Emas IMP facility whereby a record may be defined in terms of an explicit specification of component types or by cross-reference to another record should be included;
(f) Comparison of records for equality and inequality should be permitted;
(g) The type checks on record assignment statements (using "=" and "==") should be tightened in Emas IMP.

Strings

Vax IMP is a stickler for length specificatıon in all string declarations; Emas IMP is more relaxed. Certainly the need to specify a length for a constant scalar string is rather an imposition, but in all other contexts a string declaration is incomplete without a length. Emas IMP permits the use of the star specifier for functions as well as pointer variables and maps; it is not appropriate for functions any more than it is for ordinary string variables (or constant strings).

Proposals

(a) Vax IMP should permit omission of the length specifier in constant scalar string declarations;
(b) Emas IMP should require a length specifier in all otner contexts and should disallow the star specifier for functions;
(c) Emas IMP should take proper note of length specificatıon for pointer variables in type-matching and implementation;
(d) To avoid possible future problems, the use of LENGTH other than as a function should be discouraged.

Initialisation of dynamic variables

Vax IMP allows declaration statements for dynamic variables to include an initial assignment; Emas IMP does not. Although perhaps marginal, the facility is quite a convenient one, and it has the added advantage of making manifest for the variables involved that they are always assigned a value. All IMP compilers have to be able to handle run-time evaluation within declarations, so that there is no significant implementation difficulty.

Proposal

Initialisation of dynamic scalar variables should be permitted.

Scope rules

As a result of its internal organisation, the present Emas IMP compiler, unlike earlier ones, permits the programmer under certain conditions to violate the basic scope rules of IMP by using variables before they are declared. This can result in masking of errors, which are revealed on attempting to compile the program through a compiler which applies the standard rules.

Proposal

Compilers should deal with declarations in such a way that the standard scope rules are applied.

Loops

Emas IMP purports to understand what is meant by specifying that a *for* loop is to be executed a negative number of times; Vax IMP reckons that zero is the lowest valid number, and treats the negative case as an error. Emas IMP does not guarantee that on *for* loop termination the control variable is equal to the end value; Vax IMP does. The Emas approach is motivated by efficiency considerations; the Vax approach is better justified in logical terms.
Vax IMP permits a cycle introduced by a *while* clause to have an *until* clause on the associated *repeat*; Emas IMP insists on one or the other but not both. Vax IMP also permits a cycle introduced by a *for* clause to have an *until* on the associated *repeat*; again Emas does not. In this case, it seems to me that the Emas approach is better justified in terms of propriety -- refusing to complicate a simple distinction among types of loop. There could be a small loss of efficiency in cases where the programmer is obliged to use an *exit* immediately before a *repeat*, but many compilers could catch this anyway. A compromise would be to allow *until* after *while* but not after *for*, on the grounds that the latter case is a more tightly packaged concept.

Proposals

(a) the language definition should define the *for* loop in a logically consistent way;
(b) the combination of *until* clauses with other loop constructs should not be permitted.

Typographical and syntactic variations

There are a number of minor differences in the ways in which the compilers process source programs which affect the detail of what is and is not typographically acceptable -- spurious percent-signs, breaking of identifiers across lines, and the like. Where these differences affect only pathological cases, my view is that compilers should be free to make efficiency of processing the paramount consideration. However, where there is a significant loss of either convenience or error-detection, the differences need to be removed.

Emas IMP applies line continuation conventions (comma and "%C") to comments; Vax IMP does not. This can result in executable statements being ignored when a program is transferred from Vax to Emas -- which can be particularly awkward to detect. The Vax approach is simpler to apply when conventional source stream processing is used; the Emas approach presumably has advantages when special hardware stream processing facilities are exploited. At present, Emas must nonetheless make a special case of quoted strings, but with the proposals below regarding literal expressions, line-breaks in strings could be dis-allowed, so that this special case would disappear. (The choice would be less affected by implementation considerations if "%C" were less awkward to recognise; a hyphen (minus-sign) would be much simpler, and more aesthetic.)
Vax IMP allows matching finish and start atoms in a single statement (as in "finish else if symbol = 'A' start ") to be omitted. Although a minor point, it does have its own logic and its convenience is shown by the fact that is widely used.

Proposals

(a) One or the other of the approaches to continuation of comments should be adopted as standard;
(b) The dropping of "finish ... start " should be permitted;
(c) Other variations, such as the Vax IMP name function as a variant for map, should be dropped.

Operators and literal expressions

Limited character sets make it difficult to find appropriate symbols for as many operators as it would be desirable to have in the basic language. This is not a vast number (cf APL), but it is tedious and clumsy to have to use a parenthesised function notation for common cases which all the compilers treat as built-in anyway. The general availability of the full 95-character ASCII set, instead of the 64-character subset, has eased the situation and Vax IMP has elected to re-introduce an operator for modulus (absolute value) using the unambiguous vertical bar symbol instead of the rightly banished exclamation mark.
There is room for further consideration of the issue of operators v. functions. The present concept of 'intrinsic' procedures is woolly, these being defined pragmatically as those which a particular compiler picks off and implements directly, in some cases solely for efficiency, in others for more basic reasons. This leads to problems of compatibility, since there are differences in the ways intrinsic functions can be employed, compared with ordinary functions. On the other hand, they are not as good as operators (even leaving aside questions of convenience), since they cannot be used in the

construction of literal expressions. Thus the quotient of two literal values counts as a legitimate literal (operator / or //), but the remainder from the division of two literal values does not (function REM). It may be that it is necessary to consider permitting certain intrinsic functions to appear in literal expressions, though the unfortunate choice of syntax for the repetition count in literal lists may pose problems. Certainly it would be simpler all round if the need for the concept could be removed, with new operators taking the place of those intrinsics considered to be essential, and compilers hiding the other cases.

There is a particularly pressing need for an operator, or other syntactic device, as a substitute for the TOSTRING function. This is a clumsy notation for manipulating a tiny character at the best of times, but the absence of a means of incorporating control characters in literal strings is becoming increasingly awkward, and an operator for integer-to-string (ie symbol-to-string) conversion would solve the problem.

Proposals

(a) The use of vertical bars for the modulus operator should be permitted;
(b) A unary prefix operator (perhaps "$"), or some other device, should be introduced as a substitute for the TOSTRING operation, and should be valid, along with concatenation, in literal expressions.

Name arrays

Vax IMP allows the declaration and use of name arrays (as distinct from array names). Some Computer Science users strongly favour the inclusion of this facility as a language feature. My own view is that, though it is useful, it is not acceptable, since it is not capable of consistent extension. The same effect can be achieved through existing language features, by declaring records with single name-type components, and this approach extends in a controlled way to more complex forms of indirection. The objection to this approach is that it is clumsier; perhaps some consideration could be given to defining contexts in which the sole constituent of a single-component record could be denoted by the record identifier alone.

Proposal

Name arrays should not be included.

Jam-transfer assignment

The jam-transfer assignment operator is somewhat unevenly implemented, both in terms of where it is permitted and in terms of what it is understood to mean. In own and constant declarations, Vax IMP accepts it for scalar initialisation, but not for array initialisation; Emas IMP does not accept it in initialisation statements at all. Both compilers accept it for record assignment, but interpret it quite differently. Emas IMP uses it to provide the very valuable facility of record truncation, but inextricably coupled

with loss of type-checking.
The concept is a curious one, both in conceptual terms and in its implications for implementation. In regard to implementation, it implies either less work than usual by the omitting of certain checks or more work than usual by the requirement to coerce an operand to fit (the latter most obviously in the case of strings). Conceptually, it can hardly be presented as a single concept at all, and a purist approach would demand its abolition in favour of an statement of what is required, as, for example,

```
%BYTE B=K&255                    rather than %BYTE B<-K
%STRING(7) S=TRUNCATE(T,7)       rather than %STRING(7) S<-T
```

The disadvantage of the explicit approach is that it is clumsier, impossibly so for the awkward cases of coercion to signed short formats.
However, there are several considerations which make it worth considering a move in the direction of the purist approach (apart from purism). One is that it is difficult to see how any definition of jam assignment can be produced to cover the case of integers of arbitrary ranges, which will not conflict with the existing interpretations. Another is the simple consideration that not every context in which this type of capability may usefully be exercised involves an assignment operator to carry the distinction. The most obvious example is parameter passing.
There are other similar phenomena which might reasonably be included within some more general approach, such as type conversion and type aliasing. The latter is currently handled by means of the store mapping functions, as in REAL(ADDR(I)), although what is involved has little to do with store mapping and the ascent (or descent) to the address level is unfortunate.

Proposal

An analysis of what is currently covered by jam assignment and other similar operations should be carried out with a view to handling them in a more flexible and better classified fashion.

Pointer (name) variables

Both Emas IMP and Vax IMP permit initialisation of own and constant name variables, Emas in the form "= literal" and Vax in the form "== literal". The syntax is uneasy in both cases: neither takes the form of a valid assignment statement for a variable of that type. But to require that would imply permitting the store-mapping functions to appear in literal expressions, which is probably not desirable (see earlier discussion). The semantics is also dubious, with the implication that an address is nothing more than an integer value. What I have noticed about the examples of this locution which I have come across, is that the use of name variables is (literally) an indirect way of expressing what is really wanted. If on some machine the clock is at location 4 and a program requires to have a handle on it, the most direct way of doing so is to have a facility for declaring a variable of appropriate type at that location, not a name variable which points to it. An extension of the alias mechanism might provide the appropriate formalism to implement this facility, by what would be an external spec which is self-satisfying at compile-time.

In many programs which employ some kind of list processing there is a requirement to have a unique NIL value which may be name-assigned to pointer variables to represent end-of-list. Such programs presently contain their own definitions of NIL, using a variety of forms (as discussed above). It would be a distinct improvement to have NIL as a pre-defined identifier, not least because compilers could then guarantee that it is efficiently defined for that implementation and because it may, in fact, have to be defined in a way that is not representable in a standard declaration.

The role of the untyped (generic) name variable in IMP is also problematic. It has been used mainly to overcome range differences between integers and the type difference between integers and reals, but Vax IMP has pushed it further by, for example, including strings among the cases covered by the READ(NAME X) routine and offering TYPEOF and SIZEOF enquiry functions. This has the merit of attempting to force to the surface something which would otherwise be handled in a completely ad hoc fashion, but it is, in my view, misguided. The record concept in IMP opens the door to, in effect, infinitely many types, so that the idea of procedures which can handle arbitrary kinds of operand in a type-specific way becomes unrealistic. I think that the way ahead here is to encourage reserving the untyped pointer variable for operations which are not type-specific. For these, a SIZEOF function can be useful, but it would be shortsighted to define this as returning a number of bytes, rather than bits. By contrast, the way to handle genuine type differences is almost certainly through overloading of procedure identifiers. The problem of differing ranges of integers might eventually be covered by a parameter type integer (*) name, with lower and upper bounds accessible somehow.

Vax IMP does not permit a constant scalar or an element of a constant array to be used by reference (eg to be passed as a name parameter). Emas IMP does permit constant array elements, but not scalars (except as parameter to some intrinsic functions), to be so used. The restrictions, particularly those of Vax IMP, are a nuisance, and reflect a confusion in the language between the concepts of constant (invariant) and literal (explicit value), which shows itself in the circumstance that some identifiers declared as constant may be used in literal contexts but not others. It can be argued that it is desirable that the language should guarantee constancy of constants (rather than relying on hardware protection), but, of course, the fact that a parameter is passed by reference by no means always implies that the called procedure attempts to alter it. There have been suggestions to allow parameter declarations to indicate whether or not this is done, and there may be a case for re-considering these.

Granted that the indiscriminate use of pointer variables is bad programming practice, they can be used to good effect in carrying out quite low-level operations without descending to the depths. It is generally true that where the alternative is to employ the store-mapping functions in conjunction with integer addresses, the use of pointer variables is distinctly preferable, being both better controlled and more efficiently implementable on some machines -- those which use descriptors or special-purpose address registers for example. A number of facilities have been added to IMP to facilitate this use, "==" comparison for example. One addition which has been implemented in some Computer Science compilers and which is of value, is a general way of specifying a type-dependent displacement from a

pointer variable. A possible syntax is for the displacement expression to be placed in square brackets following the pointer variable: thus, for example, P[1] would denote an element of the type of P, one element on, and P[-2] would denote a similar element two elements back from P. As well as supporting general relative references, there are the obvious special cases of:

P == P[1] and P == P[-1]

Proposals

(a) Further thought should be given to the choice of syntax for name variable initialisation, and the suggestions made above regarding NIL and direct declaration via alias should be considered as ways of dispensing with a number of common cases;
(b) The language should permit constant arrays and elements of constant arrays, but not scalars, to be passed by reference;
(c) The use of untyped name variables to provide generic capability should be discouraged and a TYPEOF function should not be supported;
(d) A SIZEOF function should be supported, the unit preferably being bits;
(e) A mechanism for specifying a type-dependent displacement from a name variable should be introduced.

Arrays

Emas IMP supports multi-dimensional own and constant arrays; Vax IMP does not. This is a matter of extent of implementation and can be expected to be made good by inclusion of the facility in Vax IMP. It carries with it the implication that the order in which the elements of a multi-dimensional array are laid out must be defined as part of the language. Fortunately both the compilers follow the Fortran convention (counter-intuitive though many find it).

Proposal

Multi-dimensional own and constant arrays should be included.

Array names and maps

The forms used to declare multi-dimensional arrayname variables differ in the two implementations. Emas IMP provides array formats and a pre-defined mapping function ARRAY to allow an arbitrary part of the address space to be treated as an array; Vax IMP lacks this facility.
These differences partly reflect some indeterminacy in the IMP concept of what an array is. In effect, IMP implies that the values of the bounds and the dimensionality of an array are intrinsic properties of the array. But it does not capitalise on this to any extent, by defining enquiry functions for the bounds, for example. Particularly in a system implementation language, it can be argued that this view of arrays is too complex and inflexible. It is part of the reason why most compilers evade the issue of array mapping. Emas IMP does confront this important need, but finds itself fighting the language to provide what is required.

A more appropriate basic building block would be the concept of an array as a vector characterised by a starting position and simply a number of elements of defined type (or perhaps even a size in a defined unit). The standard IMP array declaration is then understood to decompose into two operations: declaring such an object and defining a particular mode of access to it. On this view, declaring an arrayname is seen as involving the second of these operations only. Thus, to take the easy case first, an example of a legitimate arrayname declaration would be:

```
%INTEGERARRAYNAME TABLE(1:4,1:25)
```

and a valid actual parameter for this case would be any array with number of elements known to be 100. ('Known to be' rather than just 'happening to be' for obvious advantages in checking and efficiency). Extension of this approach to the more general case, covering arrays of unknown and differing sizes, is less obvious. One possibility would be to allow the number of elements of the actual parameter to be cited as an operand in the bounds part of the arrayname declaration, or to have one unspecified lower or upper bound, as, for example:

```
%INTEGERARRAYNAME TABLE(1:4,1:£ELEMENTS//4)
  or %INTEGERARRAYNAME TABLE(1:4,1:*)
```

On this approach, assignment to the arrayname potentially involves more work than the present approach, but careful attention to the precise choice of syntax and the detail of implementation can make this small. However, it has the following advantages: arraynames are properly defined at the time of decalration; the need for the awkward construct of array formats is removed; and the array map ARRAY becomes a simple function of the two characterising properties of a vector described above -- starting position and number of elements (or size).

Proposal

The forms for specifying multi-dimensional arraynames must be standardised and the valuable facility of array mapping should be included in the language. Further consideration should be given to ways and means, perhaps using the above tentative suggestions as a starting-point.

Non-decimal numerals

Vax IMP permits real numbers to be specified using the facilities for representing numerals to bases other than 10. Peter Robertson and Ian Young have pointed out to me that this is not simply a case of pathological generality, but permits real values to be specified with a precision which may not be achievable with a decimal representation. Emas IMP has an ad hoc mechanism to cope with a particular case of the problem.

Proposal

The facility to represent real numbers in non-decimal bases should be included.

Compiler control

Both compilers have a control statement to modify the operation of the compiler in ways which are generally known only to the compiler-writer, by specifying magic numeric values. Vax IMP has an additional option statement taking a literal string as argument. This at least has the merit of presenting the argument in clear rather than code.
Emas IMP guarantees to evaluate literal comparisons at compile-time and trades on this to provide a limited form of conditional compilation. Conditional compilation is an important last resort to minimise compiler differences (as well as for other purposes), but the Emas capability is not a great deal of use in that way. It cannot be used to circumvent unwanted declarations for example or statements that are syntactically unacceptable to the compiler.

Proposal

(a) Consideration should be given to the inclusion of option as a language feature and an attempt should be made to agree standard mnemonics for the frequently required cases (suppressing checks in particular);
(b) Consideration should be given to the inclusion of a simple textually (not structurally) based conditional compilation facility.

Events

The event mechanism for signalling and trapping exceptions is a particularly valuable part of IMP, providing a capability for which there is no effective substitute in terms of other language facilities. In the nature of things, it cannot be expected that there will be complete identity of the facility as it appears in different implementations, since it must handle some highly system-dependent circumstances. There are, however, a number of minor differences of detail between the Vax and Emas implementations which could be eliminated or reduced.

Vax IMP defines a global record (or record map) EVENT, from which information relating to the event can be retrieved. Emas IMP defines a function EVENTINFO which returns two values relating to the event (combined in barbarous fashion); it also applies scope rules to this function name which, though intended to be helpful, are non-standard. The advantage of the Vax approach (and a general advantage of the use of records with respect to portability) is that it enables certain fields to be defined universally, while others are available for local use, and possible later pervasion.
Emas IMP restricts event numbers to the range 1-14, while Vax IMP provides the range 0-15, with zero corresponding to an event signalled on execution of a stop instruction. It can be useful for a calling procedure to be able to trap a stop in a called procedure, but whether the top of the range is 14 or 15 is a matter of little importance.
Vax IMP permits a trap to be specified in the form "on event *". This seems pointless, perhaps even harmful: it is hardly ever sensible to trap every event class without exception and in the rare cases which may exist it is not unreasonable to expect the list to be spelled out in full.

Proposals

(a) Event information should be provided via a record EVENT, with at least the components EVENT, SUB, EXTRA and MESSAGE being regarded as standard fields;
(b) There should be as much uniformity as possible in the choice of event numbers and sub-numbers;
(c) Wherever possible, stop should be implemented as signalling event zero;
(d) The upper limit to event numbers should be taken to be 14;
(e) The use of the star specifier in trapping events should not be permitted.

Permanent procedures

As noted earlier, the question of permanent procedures and system libraries goes beyond the matter of language compatibility into the larger issue of compatibility of the system interface. It is, however, worth making the point that, although in certain respects the distinction between features which are defined into a language and features which are expected to be provided by a permanent or system library is an important one and reflects a sensible philosophy of language design, it is nonetheless true that compatibility in terms of provision or specification of system library procedures can be just as important for portability as compatibility in language features.

Here, mention is made of only a few minor points which are known to cause fairly acute problems for portability. The kinds of differences which pose problems include:

(i) major differences of function -- clearly an extreme problem, exemplified by the situation of the most basic input/output primitives of the language, namely those which handle individual characters. Vax READ SYMBOL and PRINT SYMBOL corrrespond to Emas READ CH and PRINT CH, the two implementations having an idiosyncratic interpretation of the other pair;

(ii) systematic variation introduced by differences in system conventions. The format of file-names is an obvious, and probably inevitable, example; much more of a nuisance is the difference in the interpretation of stream numbers between Emas and Vax;

(iii) detailed differences in the typing or interpretation of arguments and results (for example, READ and the second argument for WRITE);

(iv) differences in status -- whether external, system, or 'permanent' (for example, PROMPT).

CPUTIME is a good example of a compounding of these variations:

	Emas	Vax
its status is	external	permanent
the type of its result is	real	integer
the units are	seconds	milliseconds

Of course, the virtue of the fact that these are procedures, rather than built-in language features, is that there are ways in principle to pick them up from libraries other than the system standard one. But this may imply considerable loss of efficiency if applied to very basic procedures, it is always tedious, and it is a major barrier to achieving the end of true portability -- being able to compile and run identical programs on different systems.

Proposal

A working group should be set up to examine the differences in the permanent procedures provided in the various implementations and draw up a proposed standard library.