The IMP language and compiler

P. D. Stephens

Edinburgh Regional Computing Centre, University of Edinburgh, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ

The EMAS general purpose time sharing system is notable for being coded entirely in IMP, a high level language, which was developed from Manchester University's Atlas Autocode specifically for system programming.

This paper describes the main features of the language and the implementation used for EMAS.

(Received June 1973)

The development of IMP was an integral part of the EMAS project (Whitfield and Wight, 1973) to write a multi-access operating system for the ICL 4-75 computer. The language was to be based on Atlas Autocode (Brooker, Rohl and Clark, 1966) with sufficient additions to allow all the software to be written in it without resorting to assembly language. The long term intention was to enable large parts of the EMAS system to be transported to future ranges of hardware simply by recompilation.

Atlas Autocode (AA) often seems, to those not familiar with this delightful and little known language, a curious starting point. 'Autocode' suggests, incorrectly, a low level language, while 'Atlas' implies an equally misleading machine dependence. In 1966 AA was widely used in Edinburgh University and a compiler (Bratley, Rees, Schofield and Whitfield, 1965) had been written for the University KDF9. This compiler, which was in advance of its time in that it was written entirely in Atlas Autocode, confirmed that AA was free from implementation trouble spots and reasonably suitable for system programming. Further, EMAPS was committed to supporting AA on the multi-access system, so it seemed sensible to economise in compiler writing effort by developing AA as the system programming language.

It was the intention to follow the traditions of AA as far as possible and in particular to ensure

1. That keywords continued to be self-explanatory rather than cryptic.
2. That the language remained free of implementation trouble spots.
3. That facilities requiring extensive run time support were not included.
4. That the possibility of mechanical translation of IMP to PL/I should not be excluded.

This last intention was designed (in 1966) to ensure that IMP programs and packages could be run at other installations throughout the world This laudable aim has been invalidated by the limited availability of PL/I compilers particularly on British machines.

In spite of being designed for system programming, IMP has been used extensively in Edinburgh for applications and general purpose programming.

1. The IMP language

Alphabet

The ISO (7-BIT) character set is used:

A . . . Z
a . . . z
0 . . . 9
+ − * / < ( ) = > , . : ; ' ? % @ _ & # ! ¬ ~

Names

These consist of a letter, optionally followed by more letters and/or digits in any order.

Keywords

These are underlined as in ALGOL (in this publication bold face type is used), e.g. **real**.

Blocks

An ALGOL-like block structure is used. Blocks may be nested to any depth. They may be entered only via **begin** and left only via **end.** A program is a block starting with **begin** and terminating with **endofprogram**.

Types

The principal types are **real, integer** and **string** with declarations and scope as in ALGOL. Types **real** and **integer** can be further defined by **byte, short** and **long,** subject to hardware limitations.

The length of string variables may vary, subject to a maximum length specified at declaration. This compromise gives most of the advantages of variable length strings without introducing the inefficiency of 'heap' storage.

Space for variables is normally obtained from the stack at block entry. If, however, the variables are declared with the prefix **own,** they are placed in a special area constructed at compile time and allocated at program load time. Initialisation of own variables is permitted.

Arithmetic expressions

Expressions consist of real and integer variables and constants with the operators + − * / ** ( ) in the usual way.

For example:

given **integer** i, j; **real array** $a(1:n)$; **real** x, y

then 2*x*y**2 + i*(j − 1) + a(i − 1)/3.14 is an expression.

The ALGOL integer division operator is available, represented by //.

Logical expressions

These consist of integer variables and constants and the operators:

¬	representing	logical	not
!	,,	,,	or
!!	,,	,,	exclusive or
&	,,	,,	and
<<	,,	,,	left shift
>>	,,	,,	right shift

For example:

given **integer** i, j, k

then $(i << 16)\,!\,(j << 8)\,!\,(k \,\&\, 255)$ is a valid logical expression.

Logical operations have proved very valuable in writing system software.

String expressions

A string expression consists of string variables and constants

concatenated together using . (period) as the concatenation operator.

A string constant consists of any combination of characters within quotes except that quote itself is represented by two quotes.

e.g. name . 'has not been declared'

A contextual string resolution is provided:

e.g. given **string** (25) p, q, r
$p \rightarrow q.(S).r$

The string expression S is evaluated and located within p. The portion of p before the first occurrence of S is transferred to q, and the portion after S is transferred to r. An error condition occurs if S cannot be located within p. (See also conditional instructions.)

Assignments

Assignments take the form

$$v = E$$

where v denotes any variable and E an expression.

If v is an integer variable, then E must evaluate to an integer expression (henceforth denoted by I). No implicit rounding takes place; if required, it must be explicitly requested using the built-in function provided.

If v denotes a string variable, E must be a string expression (denoted by S).

In the case of assignments to strings or to byte and short integers, a check is made that no truncation takes place on assignment. An alternative form of assignment

$$v \leftarrow E$$

suppresses the check although any resulting truncation is naturally machine dependent.

Conditional instructions

These take the form:

$$\left\{\begin{matrix}\textbf{if}\\ \textbf{unless}\end{matrix}\right\} [\textit{condition}]\ \textbf{then}\ [\textit{unconditional instrn}]$$
$$\textbf{else}\ [\textit{unconditional instrn}]$$

where the **else** clause may be omitted.

A simple condition has the form:

$$E1 \left\{\begin{matrix}=\\ \neq\\ >\\ >=\\ <\\ <=\end{matrix}\right\} E2$$

where $E1$ and $E2$ are arithmetic, logical or string expressions. String resolution (see also above) can be used as a simple condition. The condition is regarded as true if the resolution can be completed.

e.g. **if** $s \rightarrow p.(q).r$ **and** $z > 0$ **then** . . . .

A compound condition consists of a number of simple conditions linked by **and** or **or.**

There is no implied precedence between **and** and **or,** so that brackets are required to prevent ambiguity when both operators are present. Compound conditions are evaluated from left to right but only as far as is necessary for an overall verdict of true or false to be obtained.

e.g. $x = 1$ **and** $y = 0$ **and** $z < 0$
$(x > 1$ **and** $y = 0)$ **or** $z = 0$.

Unconditional instructions include:

assignments
routine calls
routine exits
jumps

A compound statement can be constructed after a condition using **start** and **finish** (not **begin** and **end**) as brackets:

e.g. **if** $x > 0$ **then start**
[list of statements]
finish

Cycles

The Atlas Autocode form of cycle is maintained:

cycle $i = I1, I2, I3$
[list of statements]
repeat

where i is an integer variable and $I1$, $I2$, $I3$ are integer expressions such that $I2 \neq 0$ and $(I3 - I1)/I2$ is an integer $>= 0$. The integer expressions are evaluated prior to entering the cycle and remain unaltered.

Two new forms of loop control have been recently introduced. Their definitions were greatly influenced by the writings of Dijkstra (1970).

The new forms of cycle are:

(*a*) **while** (condition) **cycle**
[list of statements]
repeat

(*b*) **until** (condition) **cycle**
[list of statements]
repeat

Cycling continues **while** (**until**) the condition is true. Note that **until** implies testing the condition after the body has been traversed whereas **while** implies testing before a traverse is made. Thus the body of an **until** cycle is always executed at least once, whereas the body of a **while** cycle may not be executed at all.

Cycles may be nested to any depth.

Labels and jumps

The conditions and cycles already described are designed to allow system programs to be written without requiring jumps or labels. The following facilities are provided to give compatibility with Atlas Autocode.

Labels take the form:

name:
N:
a(*N*)*:*

where N denotes an integer constant. They can only be referenced from within the block in which they are set and not within any sub-blocks. Simple labels require no declarations but vector labels require a declaration of the form **switch** *a*(*N*1: *N*2).

Jumps take the form:

→ *name*
→ *N*
→ *a*(*I*)

In the case of → *a*(*I*), I must be an integer expression and a run-time check is made that the corresponding label exists.

Structured data objects

A limited form of structured object has been introduced to facilitate the manipulation of tables within the language. The structure of such an object is described by a non-executable record format statement of the form:

record format *name* ([declaration list])
e.g. **record format** *f* (**integer** *i, j, k,* **string** (5)*s*)

All declarations (including **record**) are accepted within a record format, but arrays must have constant bounds. Space is allocated by a record declaration which may reference any previously declared format:

e.g. **record** *r*1, *r*2(*f*)
record array *r*3(1:*m*, 1:*n*)(*f*)

Records and record arrays may also be declared as own.
Elements of such tables are referenced by concatenating the record name and element name using the ISO break character (_):

e.g. *r*1_*i*
*r*3(*p*, *q*)_*s*

These compound names are acceptable in any circumstance where a simple name of the same type may be used.
Operations on complete records are restricted to record assignment including the assignment of a zero (null) record.

e.g. *r*2(*p*, *q*) = *r*1
*r*2 = 0

Routines and functions
As in ALGOL, a routine is a named block with (optional) parameters. The IMP routine heading:

{**routine** / [type] **fn**} *name* ([formal parameter list])

replaces the **begin**.
The types of routine and function [*RT*] allowed and their corresponding exit instructions are:

routine	**integerfn**	**realfn**	**stringfn**
return	**result** = *I*	**result** = *E*	**result** = *S*

As with data, a routine name must be declared before being called. This is accomplished by giving a specification of the form:

[*RT*] **spec** *name* ([formal parameter list])

It is possible to dispense with the specification if the routine is given before any reference is made to it.
The call statement is:

name ([actual parameter list])

The possible formal parameters and the corresponding actual parameters are as follows:

Formal	*Actual*
[type] **name**	
[type] **array name**	The name of an entity of the
record name	corresponding type
record array name	
[type]	An expression (*I*, *E* or *S*) of the corresponding type.
routine	A routine name
[type] **fn**	A function of corresponding type

The names associated with the formal parameter have the force of declarations within the routine body, but, in the case of *RT* parameter, a specification is also required before the formal parameter routine can be called.
An integer, real or string formal parameter is assigned at the time of call the value of actual parameter (call by value). A . . . **name** formal parameter is assigned the address of the actual parameter as evaluated at time of call (call by reference not call by substitution).

Pre-declared routines
The user's program is conceptually enclosed in a further block containing the specification and bodies of some fifty routines and functions. These routines cover mathematical and trigonometric functions, input/output and other utility routines. Naturally, in a system programming language, one does not preload routines which are not used and this is discussed further in the section on the compiler.

Input and output
This is provided by pre-declared routines—special statements are avoided. The basic routines are as follows:

select input (*I*) select output (*I*)	Arrange for subsequent input (output) to come from (go to) logical stream *I*. Mappings between logical streams and files can be made by program or (in foreground mode) by console command or (in batch mode) via the job control language.
print symbol (*I*) read symbol (*i*)	transfer symbols to (and from) the current output (input) streams converting to and from the ISO internal code. The control characters, line feed and form feed, are handled by these routines.
print string (*S*)	Outputs the string expression.
next symbol	This integer function gives the next symbol on the input stream without advancing the input pointer.

A considerable number of other routines are available for input/output of decimal and hexadecimal numbers, card images and strings. These are all written in IMP and use the basic routines described above. The compiler recognises all the basic routines and compiles a call on an Input Output Control Procedure (IOCP).
Many system programs have specialised I/O requirements. The IMP programmer has all the IMP I/O routines available to him in any situation if he supplies a suitably modified version of IOCP. The EMAS Supervisor handles its output in this manner.

Segmentation
Routines can be compiled separately provided the routine heading is prefaced by **external**. For a program or **external** routine to access an independently compiled routine, a modified specification is required of the form:

external [*RT*] **spec** ([formal parameter list])

Communication between separately compiled entities is usually via the parameter list. However, **external** variables may be declared:

e.g. **external integer** *i*
external real array *a*(1:100)

These static variables may be accessed by a number of independently compiled routines. It is possible to overlay **external** routines although on the EMAS System overlays are neither necessary nor desirable.

Pointer variables
Pointer variables may be declared as follows:

[type] **name** **record name**
[type] **array name** **record array name**

They hold the address of the entity at which they point exactly as for formal parameters of name variety. Assignment of addresses to pointer variables takes the form

p == *v*

where *p* is a pointer variable and *v* any normal variable of corresponding type. Pointer variables are often used in conjunction with mapping functions.

Store mapping
System programmers sometimes require closer control over storage than is provided by simple static and dynamic variables. A mapping function is defined by:

[type] **map** *name* ([formal parameter list])

and is similar to a normal function except that its result is

treated as an address from which a variable is fetched, or to which a variable is stored, according to the context of the call. Mapping functions thus enable symbolic names to be given to areas of storage outside the normal stack area allocated to the program. The pre-declared mapping functions:

integer (*I*) real (*I*) string (*I*)
shortinteger (*I*) longreal (*I*) record (*I*)
byteinteger (*I*)

are used to access a variable of the corresponding type whose address is given by the integer expression *I*. These functions are often used in conjunction with pointer variables.

e.g. On the ICL 4-75 the word whose address is 72 is the Channel Address Word. To access this it is necessary to code:

integername *caw*
caw = = *integer* (72)

Hereafter, any reference to caw accesses the Channel Address Word.

The function:

addr (*v*)

can be used to obtain the address of an IMP variable and the special mapping function:

array

can be used to map arrays on to data files.

The mapping of arrays and records is a powerful and widely used facility which is very similar in effect to PL/I's 'BASED STRUCTURE'.

Machine code

It is possible to write assembly code at any point within an IMP program although, naturally, this is strongly discouraged. All the instructions are available and IMP variables and labels may be used. This seemed preferable to providing special functions to permit Supervisor to use the privileged instructions.

The presence of assembly code served also to reassure those who are certain that a high level language is too 'inefficient' for a supervisor program.

Punching conventions

To facilitate punching on terminals or on cards, % character is reserved as a shift character to indicate that the following word is underlined. Statements are terminated by a semi-colon or by a newline. Consequently, if a statement is to occupy more than one line, all lines except the last are terminated by the continuation symbol **c**. Spaces and superfluous terminating characters are ignored (except within string constants). The " (double quotes) character is not used within the language and is used by EMAS Director (Rees, 1973) as a delete character when accepting input from a terminal.

Comments may be inserted by means of:

comment [text] *or* ! [text]

The Atlas Autocode fault statement:

fault [list of error conditions] → *label*

has been retained. Its effect is to intercept non-catastrophic errors and to restart the program from the specified label. If a fault occurs which has not been trapped, execution of the program ceases and a stack post mortem is output. An example of such a termination is given in Appendix 1.

The unconditional instruction

monitor

can be placed at any point in the program. Its effect is to obtain a post mortem print without otherwise disturbing execution of the program.

Language facilities withdrawn

It may be of interest to comment on three features of the original IMP specification which have been discontinued

1. *Arrays of pointer variables*

e.g. **integer name array** *na* (1:50) .

Name arrays hold the addresses of variables assigned using the address assignment operator = =. These inoffensive variables proved of little use and were dropped.

2. *Routine variables*

These were provided in an attempt to extend the routine parameter mechanism. Declarations took the form

routine name *r*
routine name array *ra* (1:*m*)

Routines could be assigned to **routine** variables

e.g. *ra*(1) = = select input
ra(2) = = select output

and finally a statement of the form

ra(*i*) (*k*)

calls the routine that was last assigned to *ra*(*i*) passing *k* as a parameter.

To enable the call to be compiled, it was necessary to restrict the routines assigned to routine variables to ones having the same parameter structure. It was further necessary to ensure that all routines assigned were global to the routine variable declarations—otherwise calls could be made on routines when their global variables were not present on the stack. These restrictions emasculated what appeared to be an interesting facility.

3. *Dynamic formats*

The early IMP compilers allowed arrays in record formats to have dynamic bounds. This meant that format statements had run time significance and that a dope-vector was required with each format. The price in execution time was judged to be too high for the advantages provided.

Compiler restrictions

The current EMAS compiler imposes two restrictions on the language described.

(*a*) The static depth of nested blocks must not exceed eleven levels of which not more than five may be **routine** . . . **end** groupings.
(*b*) Own arrays, switches, and arrays within record formats are restricted to one dimension only.

Compiler diagnostic facilities

The compiler can operate in checking or optimising mode, the former being the default. In checking mode, additional instructions are planted to ensure that:

(*a*) No variable is used before a value has been assigned to it.
(*b*) All references to array elements are within the declared bounds.
(*c*) No truncation takes place when assigning to variables of type **byte integer, short integer** or **string**.
(*d*) Overflow is tested at every stage of every arithmetic operation.
(*e*) Any cycle of the form **cycle** *i* = *p*, *q*, *r* will terminate.
(*f*) Every switch label is set.
(*g*) The source line which corresponds to the object code currently being executed is known.
(*h*) Pointers are maintained to ensure a useful post-mortem can be produced in source language terms.

Naturally, a program compiled with checks is very much larger and less efficient than the corresponding optimised program—a factor of 3 is common. Nevertheless, the checking and diagnostic facilities greatly ease the problems of debugging large programs and have proved to be one of the most valuable features of IMP.

IMP—A brief critique

The following remarks are proffered in the full knowledge that originators are often totally blind to the defects of their brain children.

It seems to us that IMP has struck a reasonable balance between what is desirable and what is possible to implement efficiently. It may err a little on the side of verbosity but it remains easy to read. Its most successful features seem to be the logical operations, strings and diagnostic facilities. Real arithmetic and routine parameters have been almost completely ignored by system programmers although both features are used by applications packages.

Arguments continue whether or not the IMP structure with its three types of bracketing (**begin . . . end, start . . . finish, cycle repeat**) produces more or less readable programs than ALGOL.

Its principal failure lies in the area of contingency handling. The fault statement is insufficiently flexible for system programmers who tend to call Director's Signal mechanism directly (Rees, 1973). A system programming language requires some form of **on** condition that permits the resumption of the interrupted program.

Purists have criticised IMP for not having a variable of type **logical**, and it is true that allowing bit operations on integer variables presumes a conversion between integers and bit representation. Nevertheless, IMP programs have been transferred between the ones-complement Univac 1108 and the twos-complement 4-75 without raising any problems in this particular area. A more serious obstacle to machine independent software is the use of mapping functions which can assume an addressing structure. However, mapping functions have proved too valuable for abolition or amendment to be possible.

The choice of the term **byte integer** was unfortunate as it causes an emotional reaction, particularly among those who dislike IBM System 360 and its architecture. A neutral term such as **character** would have been preferable.

It remains the implementer's conviction that the language would have aroused wider interest if it had been christened Implementation ALGOL or ALGOL (*I*) rather than IMP.

2. The IMP compiler

Compiler objectives

The EMAS project started two years before the delivery of the 4-75. In the meantime, a KDF9 was available for preliminary testing. The initial objectives were:

1. To provide an IMP compiler for KDF9, simulating as far as possible byte addressing and 32-bit arithmetic on a 48-bit word address machine.
2. To have a preliminary version of the 4-75 compiler ready as soon as the machine arrived.
3. To concentrate, in the first instance at least, on robust compilers with good diagnostics.

The first compilers were to be single pass to simplify the bootstrap between machines with very different device handling philosophies.

Compiler development

The starting point was the Atlas Autocode compiler for KDF9 (Bratley *et al.*, 1965). This compiler had been written entirely in Atlas Autocode. It was compiled on the Manchester Atlas and then bootstrapped to the KDF9.

The IMP compiler for KDF9 was written in the Atlas Autocode compatible subset of IMP and simulated byte addressing and 32-bit arithmetic by extensive use of subroutines. Arithmetic operations were 2-4 times slower than in Atlas Autocode and array access some 6 times slower. However, the bootstrapped compiler was only about 50 per cent slower than the Atlas Autocode compiler, thus showing what a small portion of compiling time is spent in arithmetic operations.

For System 4, the compiler was bootstrapped from the KDF9 compiler via assembly language (due to difficulties with the manufacturers early operating systems). This compiler was also bootstrapped onto the IBM 360/50 (Yarwood, 1970).

The early EMAS components were compiled on the manufacturers J level operating system and transferred to the EMAS system as binary magnetic tape. As soon as the supervisor was sufficiently robust, the IMP compiler followed this by now well worn route.

Since its establishment on System 4, the IMP compiler has been bootstrapped through itself seven times. Two of these were due to changes in register conventions and sub-system standards inevitable in the 'iterative' design of complex software. One was due to hand-coding of several routines to produce faster compiling times. (A 20 per cent increase in speed was obtained after the usual difficulties of debugging assembler coding had wasted several months.) The remaining four bootstraps were to enable improvements to be made in the following areas:

Release 5	routine entry and exit	effort = three months
Release 6	register allocation	effort = two months
Release 7	expression optimisation	effort = six months
Release 8	simple loop optimisation	effort = five months

Of these, the register allocation required the fewest source statements and produced the biggest improvement in object code. The expression optimisation required the most new source code and produced the least effect.

The effect of these changes is that the Release 8 compiler produces about half the amount of object code that Release 1 produced for the same program. Release 8 object code is rather more than twice as fast as Release 1 object code. Since the compiler is written in IMP, compiling speeds have increased similarly. These figures enable the four months spent hand coding analysis routines to be seen as the waste of effort it undoubtedly was.

Object program

The object program produced by the compiler consists of three areas. A shareable area of code, constants and symbol tables, a non-shareable area of own variables and external references, and lastly some linkage data. The object program format is described elsewhere (Millard and Whitfield, 1973). Note, however that EMAS program sharing (Whitfield and Wight, 1973) demands that a shared program executes correctly even if it is at a different address in virtual memory. This causes the compiler writer some problems—for example by preventing the use of address constants—although the effect on object code efficiency is small.

The IMP object program uses a stack for variables and temporary space. Each routine claims its 'stack frame' on entry and releases it on exit. A 'display' points to the stack frames available at any instant as defined by the scope rules. In IMP, this display is kept in the general registers, thus making local variables directly addressable at all times. This leads to efficient object code except when passing routines as parameters, but requires a limit to be placed on the depth of nested blocks.

On a routine call the parameters are placed ahead of the stack top such that they will appear in the local variable space of the called routine when it claims its stack frame.

One register of the object program is reserved to address 'PERM'. This is a small collection of assembly code routines provided to carry out basic services for the object program, e.g. dynamic array declaration. A checked program requires some 25 routines (3660 bytes)—an optimised program only six routines (1560 bytes). The main constituent of PERM is a large table of multiples of 4096 which are required to overcome the addressing limitation of System 360/System 4 architecture.

This addressing limitation also requires that a routine's stack frame may have to be divided into two parts. The first part, restricted to 4096 bytes in length, contains scalar variables and pointers to elements in the second part. The second part contains strings, records and arrays. If any arrays have dynamic bounds, the size of the second part of the frame will not be known until run time. Frequently the size of the first part is substantially less than 4096 bytes, in which case strings, records and any arrays with constant bounds are transferred from part 2 to part 1 to utilise the balance of the space and obviate the inefficiencies caused by the use of pointers.

The pre-declared routines are divided into two groups. The 'intrinsic' routines are sufficiently trivial for in-line code to be generated. The remainder are provided by external routines and functions written in IMP. When a pre-declared name is first referenced, the compiler generates the appropriate linkage data to enable the routine to be included at program load time. A systematic change of name is required to avoid confusion between pre-declared routines and user written external routines. This arrangement has the considerable advantage of enabling much of the language support software to be written in IMP and thus to be available to all the IMP compilers. It also avoids including unwanted material with system programs.

Compiler structure

The internal structure of the compiler is a field of very limited interest. Some features of more general interest are described below.

The compiler is a normal IMP program obtaining its input via the read symbol routine and outputting a program listing and error messages via the print symbol routine. The compiler hands its binary in small chunks to a routine provided at compiler load time. This routine constructs the output file. By varying the binary output routine, the output can be an EMAS file, binary cards or a file for some other operating regime. In particular, an output routine exists which puts the code directly into core and executes it—giving a 'student crunching' system using the standard shared compiler. This arrangement is not the most efficient imaginable, but it does have the attraction that the compiler can be operated in any (sufficiently large) machine possessing an IMP system and opens the door to a variety of bootstrapping techniques.

The compiling technique is one-pass, multiphase:

Phase 1 Input of the source program and construction of the listing.

Phase 2 Syntactical Analysis using a table-driven variation of the method of recursive descent. The syntax tables are generated automatically. This phase also includes all the dictionary building.

Phase 3 Compilation proper—the following subphases are applied as necessary:

- 3*a* Generation of an internal representation of all assignments, expressions and jumps.
- 3*b* Examination of the internal form for machine-independent optimisations.
- 3*c* Examination of the internal form for shortcuts in the object code (Object machine dependent).
- 3*d* Allocation of registers (Object machine dependent).
- 3*e* Generation of the binary (Object machine dependent).

Phase 4 Synthesis of the object program from small pieces of binary.

Phase 1 is applied to the source program, a line at a time; the other phases, a statement at a time. **Table 1** gives the CPU times spent in each phase for two sample programs.

Table 1

				PROGRAM		
				1	2	3
Source file (statements)				3839	261	261
Object code (bytes)				34944	4616	11008
Total cpu time (seconds)				80·15	8·777	8·101
Total no. of page turns*				2205	607	634
Break down per Phases						
Phase 1	percentages of total		cpu	17·38	15·11	16·37
	,,	,,	p turns	33·06	58·12	55·68
Phase 2	,,	,,	cpu	31·03	21·02	20·37
	,,	,,	p turns	11·93	2·80	2·68
Phase 3	,,	,,	cpu	50·80	61·54	62·33
	,,	,,	p turns	48·66	22·90	10·78
Phase 4	,,	,,	cpu	0·79	2·34	9·13
			p turns	6·35	16·14	28·86

Program 1 is the principal component of the current EMAS supervisor (optimised compilation)

Program 2 is a matrix program—predominantly performing real arithmetic on two dimensional arrays (optimised compilation)

Program 3 is program 2—checked compilation.

*The EMAS supervisor charges for a page turn whenever it brings a page into core on behalf of a process. Currently the charge is equivalent to 0·003 seconds of cpu time.

Every effort has been made to isolate those parts of the compiler which are dependent on the characteristics of the target machine. It is distressing how large phase 3*c* has to be to utilise fully the 360 order code, in particular the store-to-store and store-immediate instructions.

The compiler makes no attempt to improve the paging characteristics of the object program in spite of work done at Edinburgh showing how effective rearrangement can be (Pavelin, 1970). However, unnecessary branching is avoided, and in-line code preferred to subroutines except for the following operations:

Initialisation (execution of the first **begin**)
Termination
Execution of a fault statement
Dynamic array declarations (actually two subroutines)
String Resolution

In-line code is produced for exponentiation, string concatenation and fix/float operations.

Optimisation techniques

In optimising a source program, the 'window' technique is used: Phases 1 and 2 are allowed to run ahead of Phase 3 and a cyclic buffer is used to store up to 15 analysed statements. When elementary examination discovers a group of statements that must be executed in sequence, Phase 3 is applied to the group of statements. This technique is adequate to permit optimisation of most loops as can be seen from Appendix 2.

When designing the optimisation, the (highly dubious?) assumption was made that experienced system programmers would code to a high standard. The compiler makes no attempt

to perform optimisations that can be better done by the programmer. No attempt is made to move constant operations out of loops, and common sub-expression optimising is restricted to expressions which cannot be further simplified by the programmer.

Bootstrapping

Compilers have been produced which implement substantial subsets of IMP on PDP8, PDP9, PDP11, PDP15, Modular 1 and Univac 1108 machines.

The easiest way to produce an IMP compiler for a new machine requires EMAS or some other large IMP system. The most suitable existing IMP compiler is altered to produce object code for the new machine. This compiler compiles itself and its supporting input/output routines using a binary output routine that produces cards suitable for loading onto the new machine. Eventually this new compiler compiles itself on the new machine to produce a self-supporting IMP compiler.

Where a suitable large IMP system is not available, the SKIMP bootstrap method is used. SKIMP is a subset of IMP (roughly the Atlas Autocode compatible subset), and a compiler exists written in SKIMP to produce hypothetical assembly code (HAL) for an austere, one accumulator, three index register machine. Most of the compiler support material is written in SKIMP and exists in source and compiled (HAL) form. HAL has been designed to be assembled by most current macro assemblers. To implement SKIMP on a new machine, it is necessary to write the macros to enable the HAL version of the SKIMP compiler to be assembled, and also to write a small amount of input/output software in machine code. Once the HAL version is operational, it can be used to bootstrap an orthodox compiler or to improve itself by iteration. The IMP compiler for Univac 1108 was produced via this route in less than six months.

The efficiency' of IMP

There is no doubt that EMAS has gained immensely by being written in IMP, yet the question often asked by visiting system programmers is 'How efficient is IMP?' This question is not easy to answer.

Two routines in EMAS and twenty in the compiler have been hand-coded. The gains in performance or reductions in size have varied from an encouraging 2 per cent to a rather discouraging 40 per cent. The majority have fallen into the 10 per cent to 20 per cent range. One routine in EMAS—the interrupt analysis routine—was originally written in assembler code. This routine has recently been rewritten in IMP and this time the IMP version is smaller by 11 per cent and presumably faster by a like amount. It is probably fair to conclude that the IMP compiler produces code about as efficient as assembler written by programmers under the usual pressures. Both of these fall short of the optimum possible.

In the matter of variable space, the advantage seems to lie with IMP. The stack system economises on storage by allocating it only to those routines currently active. There is no reason why assembler programmers should not use a stack, but in practice they usually prefer to allocate private storage to each subroutine. The total local variable space required by the routines of the current EMAS supervisor is 17848 bytes, but a stack of 3000 bytes has proved adequate, giving a substantial saving of 14848 bytes. For paged software, the stack system has the added attraction of economising on page faults.

Programming effort

The work described in this paper took the author about four years spread over the years 1967-1972 inclusive. The production of the 1108 compiler took about six months. It is estimated that a tolerable compiler could be produced for any conventional machine in about the same time. A further period might be required to improve the object code to the standard of the current EMAS compiler, but much would depend on the order code of the new machine.

On system programming in IMP

All members of the EMAS project agree that working in a high level language was a great advantage. The volume of coding was greatly reduced—a listing of all the system source code will fit in an average briefcase and still leave room for sandwiches. This meant that the programmers working on the system could be reasonably familiar with all the code, not just their own section. Consequently, a system crash could be diagnosed and solved by one programmer rather than a committee. The checking facilities pinpointed many (but alas not all!) coding errors before they appeared as mysterious, transient bugs. The run-time diagnostics were valuable to the subsystem writers, although of considerably less value in detecting an error in, for example, the page fault routine.

Probably the most valuable feature of IMP was the encouragement it gave to structured programming. Within the structure, an unsatisfactory routine or component could be identified, redesigned and recoded in a short time, without disturbing the rest of the system.

The EMAS programmers who had previous experience of high-level languages adapted easily to system programming in IMP. They produced compact, highly structured programs which were easy to maintain or amend despite defects in commentary and/or documentation. They seldom worried about the efficiency of object code produced by the compiler, but their programs generally performed well. This group included the most productive programmers working on the project. Programmers with a background of assembly languages were less happy with IMP and seldom used its more advanced features such as recursion. They produced well commented and documented programs that nevertheless proved difficult to maintain since they lacked structure. This group worried about the efficiency of object code produced by the compiler to the extent of examining the listings of code produced, yet their programs were often large in size and slow in execution. Some of the least productive programmers were included in this group.

In view of our experiences with IMP, it is demoralising to thumb through the 'situations vacant' columns of the newspapers and read that one has no chance of being recruited to write the 'software of the future' without 'several years experience of assembly language programming, preferably on an IBM 360'!

Acknowledgements

The debt to the designers and implementers of Atlas Autocode is as large as it is obvious. Particular credit is due to P. Bratley, D. Rees, P. Schofield and H. Whitfield who wrote the Atlas Autocode compiler for KDF9.

H. Dewar wrote the IMP compiler for the PDP9 and PDP15 machines while S. Hayes, N. Shelness and K. Yarwood contributed IMP compilers for the PDP8, Modular 1 and PDP11 respectively. The SKIMP/HAL bootstrapping method was developed from a teaching project designed by D. J. Rees.

The above, and many others too numerous to mention individually, contributed ideas and suggestions, or joined in the heavy but good-natured criticism with which innovations were invariably received.

Appendix 1

The following example shows the diagnostics given after an array bound exceeded fault with the given program and data.

```
Program
 1    %BEGIN
 2      %INTEGER GIVEN,SQ
 3      %INTEGERFNSPEC MAX FACTOR(%INTEGER N)
```

```
4       !
5       !FINDS ALL PRIME NUMBERS LESS THAN GIVEN
6       !
7       !METHOD IS TO FILL ODD LOCATIONS IN % C
         ARRAY A WITH ONE AND
8       !THEN DELETE MULTIPLES OF ALL POSSIBLE % C
         FACTORS
9       !
10      READ(GIVEN);  !OBTAIN DATA
11      !
12      %BEGIN
13        %INTEGER I, SQ, TAB
14        %BYTEINTEGERARRAY A(2:GIVEN - 1)
15        A(2) = 1;  !SPECIAL CASE ONLY EVEN PRIME
16        %CYCLE I = 3, 1, GIVEN - 1
17          %IF I&1 = 0 %THEN A(I) = 0 %ELSE A(I) = 1
18        %REPEAT
19        !
20        I = 3
21        %WHILE I < MAX FACTOR(GIVEN) %CYCLE
22          %IF A(I)#0 %THEN %START
23            SQ = I**2
24            %WHILE SQ < GIVEN %CYCLE
25              A(SQ) = 0;  !THIS NUMBER NOT PRIME
26              SQ = SQ + 2*I
27            %REPEAT
28          %FINISH
29          I = I + 2
30        %REPEAT
31        !
32        !NOW PRINT OUT ANSWERS AT TEN % C
          TO A LINE
33        !
34        PRINTSTRING('
35'       PRIMES LESS THAN ')
36        WRITE(GIVEN,4)
37        NEWLINES(2)
38        !
39        TAB = 0
40        %CYCLE I = 2, 1, GIVEN; % C
                    !ERROR. SHOULD BE 'GIVEN - 1'
41          %IF A(I)#0 %THEN %START
42            WRITE(I, 5)
43            TAB = TAB + 1
44            %IF TAB = 10 %THEN %START
45              TAB = 0
46              NEWLINE
47            %FINISH
48          %FINISH
49        %REPEAT
50        %END;  !OF INNER BLOCK
51      %INTEGERFN MAX FACTOR(%INTEGER N)
52        !
53        !RETURNS ANSWER SUCH THAT % C
           (ANSWER - 1)**2 < = N < ANSWER**2
54        !
55        %INTEGER I
56        I = 1
57        %WHILE I**2 < = N %THEN I = I + 1
58        %RESULT = I
59        %END; !OF INTEGER FUNCTION MAX FACTOR
60      %ENDOFPROGRAM
```

When this program is run with a datum of 99 the following output is obtained.

```
PRIMES LESS THAN 99
 2    3    5    7   11   13   17   19   23   29
31   37   41   43   47   53   59   61   67   71
73   79   83   89   97
MONITOR ENTERED FROM IMP
ARRAY BOUND FAULT 99
ENTERED FROM LINE 41 OF BLOCK STARTING AT LINE 12
LOCAL VARIABLES
TAB = 5
SQ = 105
I = 99
ENTERED FROM LINE 12 OF BLOCK STARTING AT LINE 1
LOCAL VARIABLES
SQ = NOT ASSIGNED
GIVEN = 99
STOPPED AT LINE 41
```

Appendix 2

The object code produced for an optimising compilation of a piece of IMP.

```
given           integer i, j, k, n
                integer array a(0:100), b(0:n)
then the code produced for
                cycle i = 1, 1, n
                      j = j + a(i)
                      k = k + b(i + 1)
                repeat
is as follows
                USING STACKFRAME, 9
                USING CODEBASE, 10
*PROLOGUE       TO THE CYCLE
                L   4,AB0 SET GR4 TO POINT AT
                          DYNAMIC ARRAY B
                L   5,K   ASSIGN K TO GR5
                LA  6,1   ASSIGN INCREMENT TO GR6
                L   7,N   AND FINAL VALUE(N) TO GR7
                L   8,J   ASSIGN J TO GR8
                LA  1,1   ASSIGN I TO GR1 AND SET TO 1
*END OF PROLOGUE=START OF CYCLE BODY
CYCO1           LR  2,1
                SLL 2,2   SET GR2 = 4*I
                A   8,A(2) J = J + A(I) STATIC ARRAY A
*                         IS IN STACKFRAME (UNLIKE B)
                A   5,4(2,4)  K = K + B(I + 1)
                BXLE 1, 6, CYCO1 REPEAT
*CYCLE EPILOGUE ASSUMING ALL REGISTERS NEED TO
*BE UNSET
*I.E. NEXT STATEMENT IS BRANCH OR ROUTINE CALL.
                ST  5,K   RETURN K TO STORE
                ST  8,J   RETURN J TO STORE
                ST  7,I   SET I TO FINAL VALUE
```

This code is not the optimum possible since it is possible to rearrange the cycle so that it is of the form

cycle $i = 4, 4, 4 * n$

this saves a LR and SLL in the innerloop at the expense of more in the prologue and epilogue.

References

BRATLEY, P., REES, D. J., SCHOFIELD, P. D. A., and WHITFIELD, H. (1965). *Atlas Autocode Compiler for KDF9*, Edinburgh University Computer Unit Report No. 4.

BROOKER, R. A., ROHL, J. S., and CLARK, S. R. (1966). The main feature of Atlas Autocode, *The Computer Journal*, Vol. 8, pp. 303-310.

DIJKSTRA, E. W. (1970). *Notes on Structural Programming*, Technical University of Eindhoven Report No. 70-WSK-03.

MILLARD, G. E., REES, D. J., and WHITFIELD, H. (1973). The Standard EMAS Subsystem, *The Computer Journal* (to be published).

PAVELIN, C. J. (1970). *The improvement of program behaviour in paged computer systems*, Edinburgh University Ph.D. Thesis.

REES, D. J. (1973). The EMAS Director, *The Computer Journal* (to be published).

WHITFIELD, H., and WIGHT, A. S. (1973). EMAS—The Edinburgh Multi-Access System, *The Computer Journal*, Vol. 16, No. 4, pp. 331-346.

YARWOOD, J. K. (1970). Towards machine independent processors. *The Computer Bulletin*, Vol. 14, No. 7, pp. 219-221.