M. Alfonseca
M. L. Tavera

A Machine-Independent APL Interpreter

Abstract: The problem of writing machine-independent APL interpreters is solved by means of a systems programming approach making use of an intermediate level language specially designed for that purpose. This paper describes the language, as well as the procedure used to build universal interpreters. Three compilers that translate this language for three different machines have been written so far, and an APL interpreter has been finished.

Introduction

When a new computer is developed, it generally has its own machine and assembly languages, usually different from those of other machines; most software, including high level language translators, must be rewritten for the new machine. The cost of this work would be greatly reduced if some of the software could be made machine-independent.

One high level language not commonly provided with new machines is APL [1]. This highly sophisticated interpretive language includes a large number of symbolic built-in functions (primitive functions) and operators that render it possible to write complicated programs in a concise form with a simple syntax.

Primitive APL functions and operators take arrays as well as scalars as their working objects, so that loopless programs may be written. Thus, good APL programming somewhat counteracts the loss in translation time inherent in interpretive systems compared to compiling systems. Besides, many of the common array-handling operations, such as matrix products, matrix inversions, and so forth, are primitives in the language.

A universal APL interpreter which would make this language available on many machines would be welcome, especially in view of the fact that use of the language is growing.

As an example of this need, when the IBM System/7 (a sensor-based computer) was first announced, it was provided with only a disk support system and a primitive assembler. More complete software was added later, including a FORTRAN compiler. We were interested in being able to manage the System/7 sensors by means of APL. We could not simply use one of the different APL systems available for the IBM System/370, because the assembly languages and architectures of the systems are different. Therefore, we built a System/7 APL interpreter, written in assembly language [2].

A new sensor-based computer, the IBM Series/1 [3], has recently been announced as an alternative for System/7. If we want to use the programs we wrote for the System/7 on the new computer, another APL interpreter will have to be written, because the assembly languages and architectures are again different.

Instead of building a Series/1 APL interpreter, and probably having to face the same problem again in the future, we decided to try to write a universal APL interpreter, as independent as possible from the machine.

APL system requirements

An interpretive APL system should have the following properties [4]:

1. Time sharing should be provided for, so that different users may have access to it at the same time by means of terminals. Each user is assigned a section of main storage, called an active work space, where he may keep and execute his data and APL functions. He is also assigned a library in auxiliary memory where he may store copies of his active work spaces. The available main memory is usually split up into several active work spaces (slots). At a given moment, the number of users connected to the system may be greater than the number of slots. In this case, copies of all

active work spaces are kept in an auxiliary file, and whenever a user is given control, his active work space is swapped into one of the memory slots.

2. The data contained in a work space may be internally represented as one of the following types:

   Boolean, occupying one bit per element of data,
   Integer, one, two, or four bytes per element,
   Floating point, typically eight bytes per element,
   Literal, one byte per element,
   Pointer, the value of which is an address.

   Memory allocation within the work space is dynamic; some garbage collection procedure should be provided. Memory management requires an extensive use of pointers.

3. The system is usually made up of the following parts:

   A supervisor, which manages the time sharing and the terminal and disk input/output operations.
   An interpreter, which must be reentrant, i.e., all modifications should be done only in the work space.

   The supervisor is in itself machine-dependent and a general operating system or subsystem has often been used as such [5]. However, this is not the case with the interpreter, where only slight details, like paper width or floating point precision limits, may have to be varied with the machine.

Therefore, we decided to build a machine-independent APL interpreter with a machine-dependent supervisor to be added for each particular implementation.

Selection of the interpreter writing language

Having decided to write a universal APL interpreter, we then had to face the problem of choosing the language we would use to write it. Three criteria were considered: the degree of machine independence of the language, the extent to which the APL system requirements described above could be met, and finally the performance of the APL interpreter in terms of both execution efficiency and minimal storage requirements.

Since interpreters are usually about an order of magnitude slower than compilers for the same language, the process of construction of our interpreter should not introduce a noticeable degradation in its execution speed. On the other hand, since we intend to make the system available for both minicomputers and mainframes, it should be as small as possible. Since these two conditions are frequently opposed to each other, an optimal solution for both cannot be met. We are thus prepared to trade off slight losses in execution speed for significant reductions in size.

Four different language classes were considered.

- *Assembly languages*

Assembly languages are the best suited to meet all the system requirements; they allow maximum flexibility to the system programmer, while providing the best size and speed. However, they are completely inadequate for meeting the independence criterion, since every machine uses its own assembly language.

- *Macro language*

A first approach to getting machine independence would be to design a general macro language. Each macroinstruction would be generated by means of the macro definition facilities provided in most assembly languages.

One advantage of this approach is that flexibility of the language would be only slightly less than that of assembly languages. On the other hand, machine independence cannot be completely assured, because not all assemblers are macroassemblers (as was the case with the first version of the System/7 assembler), and not all of these have the same power. There is a danger that either the number of macroinstructions which must be defined will grow too large or that each macro will become too complex. For instance, suppose we want to define one or several macros to add two or three arguments. Since we may have to add arguments of different types (recall requirement 2), there are two possible solutions to this problem: define one macro for every possible combination and for each number of arguments to be added (at least 30 macros would be needed just for the addition operation); define a single highly complex macro which would combine all possible cases. The macroassembler would then have to provide conditional macroinstructions and the ability to ascertain the existence of an argument at preassembly time.

- *High level languages*

A second approach would be the use of an existing high level language as the interpreter writing language. Ideally, there would be a good, high level programming language and, for each machine, a compiler to translate this language into efficient machine code. However, this situation does not exist yet, and current high level languages, while highly readable and capable of providing concise programs, add both to size and execution time due to the compilation process and the run-time environment. Two high level languages were considered:

1. FORTRAN is a widely used high level language which assures some degree of machine independence, because compilers for different machines differ only in minor details. However, the limited flexibility of this language makes it difficult to meet some of the APL system requirements. In particular, FORTRAN data representation does not allow easy management of Bool-

ean data or of integers occupying only one byte. The required Boolean operations would include logical AND, OR, exclusive OR, and negation of bit strings, plus selection and testing of individual bits or groups of bits. More important is the fact that FORTRAN compilers cannot usually generate reentrant code, a necessary condition for writing a time sharing system.

2. PL/I is another commonly used high level language, and compilers for it are provided for most machines. It is a more flexible language than FORTRAN, and data types are reasonably well managed. Reentrant code generation can be selected as an option. A drawback is that PL/I compilers for different machines usually implement different subsets of the language. Thus, should we select a given subset, there is no guarantee that the one implemented in a new machine will contain all the features we have selected. The PL/I compiler provided with the machine might have to be extended to meet our requirements.

• *Systems programming approach*

The systems programming approach consists in the use of a language higher than assembly language but lower than high level languages as the systems programming language. Assembler languages are obviously the most flexible and efficient, while high level languages give the maximum machine independence and readability. Systems programming languages usually combine the properties of both in the sense that they provide the option of including built-in functions and assembly language statements within the high level environment. They are also provided with good optimizing compilers which produce very efficient code.

Standard systems programming languages, however, can only be used at the expense of a loss in machine independence, because of their machine language features, which are obviously dependent on the computer. In addition, some of the most widely used systems programming languages do not manage floating point data, obviously necessary to write an APL interpreter.

Even a subset of an existing systems programming language would not be an optimal solution to our threefold problem of machine independence, flexibility, and efficiency. In the first place, the language would have to be stripped of some of its flexible features to assure machine independence. Thus, such languages would again become high level languages, discussed in the preceding paragraph. In the second place, the semantics of these languages, usually PL/I- or ALGOL-like, would make it difficult and time-consuming to build good optimizing compilers. For these reasons, to solve our specific problem, we decided to design our own ad hoc systems programming language. It should be really intermediate in the sense that it should have the semantics of assembly languages but with a higher level syntax, and it should be as easy to analyze as possible, with an eye to reducing the programming effort required to build compilers that produce highly efficient code. We shall call such a language an "intermediate language (IL)."

The procedure followed to design the IL instructions was to select the most common operations in the assembly languages of different IBM machines [6] and to represent them with a high level syntax. Instructions not arising naturally from the assembly level, such as IF-THEN-ELSE, DO, and so forth, are not a part of IL because our objective was only to define a substitute for assembly languages; we were not concerned with high level language properties such as complex operations and those making structured programming easier. Also, special instructions such as BXLE, TR (System/370 assembler) appearing in a few assembly languages have not been selected, to safeguard machine independence.

The language

The only assumption about the machine in which IL may eventually be implemented is that its memory is considered to be a vector of units of fixed but not defined size, consecutively numbered. Appendix A shows the syntax of IL.

• *Data objects*

The data objects of the language are numeric constants (fixed point integers and floating or decimal rational numbers) and identifiers which may name different types of data: four types of variables; pointers; labels; routine names; and parameters.

A variable has four different attributes:

The memory address associated with it,
Its type,
Its length (number of elements),
The actual values of the elements.

The address of the variable defines the location of the first memory unit of the space allocated to the variable.

Variable values may be integer or rational. Integer values may be of three different types, corresponding to the assignment of one, two, or four memory units per element.

A variable may contain one or several elements in a linear structure. The fact that we have defined IL semantics to be as close as possible to machine language level precludes the inclusion of more complex structures (matrices, lists, etc.) which must always be ultimately represented in a linear memory.

A pointer is a name the associated value of which is considered to be the address of some variable. In most cases a register would be assigned to it, although memory locations may also be used.

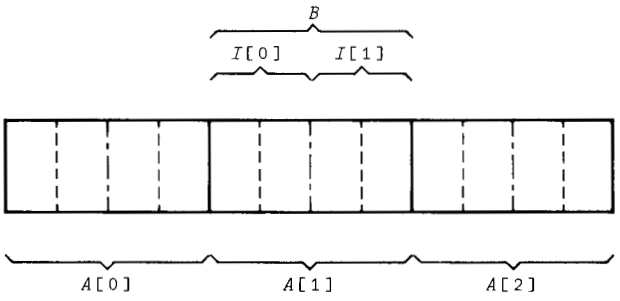

Figure 1 Space allocation for example 1.

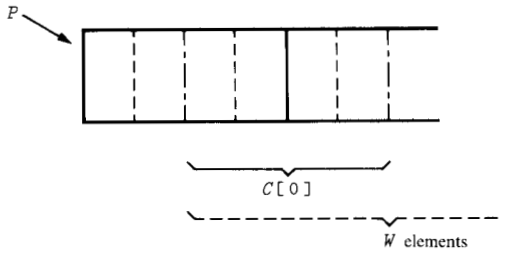

Figure 2 Space allocation for example 2.

Labels are names that may be associated with instructions, whereas routine names are entry points to the different subprograms which may make up a complete program. A subprogram may have several entry points.

A parameter is an identifier the value of which cannot be changed during program execution.

The first letter of the name assigned to an identifier implicitly declares its type. Appendix B shows the conventions used.

● *Program organization*

An IL program consists of two different parts: declarations and executable statements, separated by a separation statement.

All variables appearing in a program must be declared. This can be done in either of the following ways:

1. Assigning one or more initial values to the variable name. The length of the variable is thus defined as the number of values given. The syntax of these statements is

   variable name ← *values*

2. Assigning a synonym to a variable name previously declared. This feature permits the declaration of variables with undefined attributes, allowing the following possibilities:

   a. Variables of different type, sharing the same address,
   b. Variables of undefined length,
   c. Variables of undefined address, depending on the value of a pointer on which the variable is based. The address assigned to the variable is computed as the addition of the value of the pointer plus an offset.

All preceding possibilities are mutually compatible and can be combined in a single statement with the following syntax:

variable-name optional-index = variable-name optional-index

Example 1 (see Fig. 1)

```
A ← 1 2 3
B=A[1]
I[2]=B
```

Variable A is implicitly defined by its first letter as an integer variable with four memory units per element. Its length is three, and the initial values of its elements are 1, 2, and 3. Variable B is defined as a single element variable with the same address as the second element of A (indexing uses origin zero). Variable I is declared as an integer variable with two memory units per element, of length two, and the address of which is the same as that of B.

Example 2 (see Fig. 2)

```
W←4
C[W]=P[2]
```

Variable W is defined as a single element integer variable occupying a single memory unit and with an initial value of 4.

Variable C is declared as an integer, four-unit-per-element variable the address of which is offset two units from that pointed to by pointer P; its length varies with the value of W.

Executable IL statements are analyzed from right to left. Functions are executed without special precedence rules, in the order they are found; parentheses are not allowed. In addition to the assignment and the standard arithmetic operations, the following functions are allowed:

Pointing, $P \rightarrow X$, assigns to pointer P the address of variable X.
Incrementing, $P \Delta X$, increments the value of pointer P by the value of X.
Shifting, $A \uparrow B$, shifts the value of B to the left A bits (this operation is equivalent to a multiplication of B by the A power of 2), while $A \downarrow B$ shifts B to the right.

Logical bit to bit operations, ∨ (inclusive or), ∧ (and), ⊛ (exclusive or), ~ (not). They operate on values of any integer type on a bit to bit basis.

Data objects of any integer type and pointers can be freely mixed in the IL statements. The assignment instruction also allows for conversion of integer to rational data and vice versa.

Example 3

$F \leftarrow P + I \times A$

Primitive operations only affect data objects of length one, including indexed variables and pointers. The assignment statement, $A \leftarrow B$, is again an exception, in the sense that if A is a variable of length different from one, the required number of memory units is copied, one unit at a time, from the address of B to the address of A in ascending order (the unit with the lowest address is copied first).

The following transfer instructions have been included: unconditional transfer, $\rightarrow E$, corresponding to the unconditional branch in most machine languages; conditional transfer, $\rightarrow F \; IF \; CONDITION$, corresponding to the conditional branch; test bits; $\rightarrow E \; IF \; V \wedge I$, corresponding to the test under mask instruction and meaning that the transfer is taken if the "logical and" of V and I is not null.

IL does not contain special input/output instructions. The reason is that all APL input/output operations are managed by the supervisor and, whenever the interpreter needs one, it calls a supervisor subroutine.

In an IL instruction, the symbol ⍝ indicates that everything at its right up to the end of the line is a comment and should be ignored.

General procedure

The procedure for building a universal APL interpreter using IL as a systems programming language is accomplished according to the following scheme:

An APL interpreter is written in IL.
A compiler is built that translates IL programs into assembly language for machine M_1.
The interpreter is compiled. The final product is an APL interpreter directly executable on machine M_1.

This procedure is displayed in Fig. 3, where square boxes represent APL interpreters written in the language at the bottom. The T-like figure represents a compiler written in the language at the bottom and translating the language at the left into the language at the right.

We have chosen APL as the language for writing the compiler, in spite of the loss in efficiency inherent in any interpretive language, because compiler performance is not important at all in this environment; once the APL/IL interpreter has been written, it must be compiled only once for each machine. Besides that, APL is a very suitable language for writing compilers quickly [7, 8].

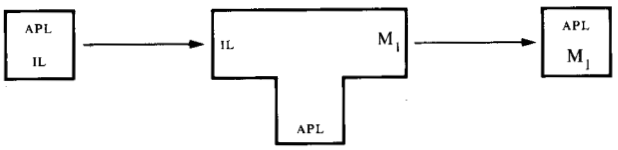

Figure 3 General procedure for building a universal APL interpreter.

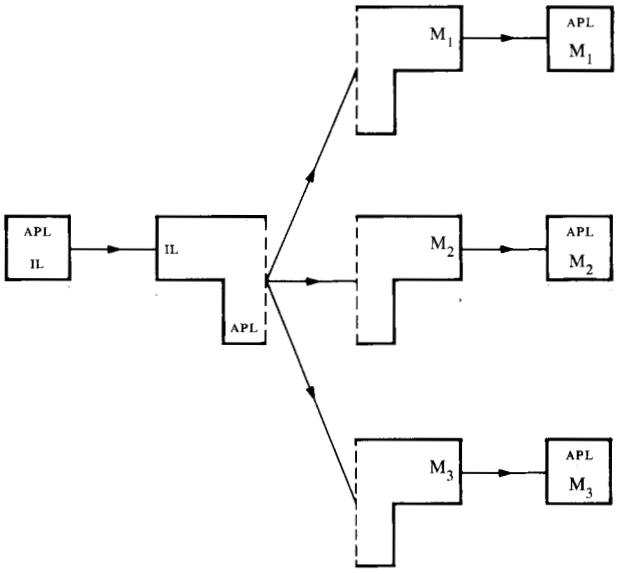

Figure 4 Application of the general procedure to three different cases.

To obtain an APL interpreter directly executable on a different machine, M_2, only the code generator of the compiler need be rewritten.

The compilers can be executed on any base machine where APL is available. We are presently using APLSV on a System/370, but the base machine can be changed at any time with no further cost.

Let us consider three machines M_1, M_2, and M_3. With our procedure one interpreter and three compilers must be programmed in order to implement APL on all of them (see Fig. 4).

Efficiency of the procedure

Suppose we intend to implement an APL interpreter on n different machines. Here we compare the effort by the programmer, the amount of space required in storage, and the execution speed of three different approaches.

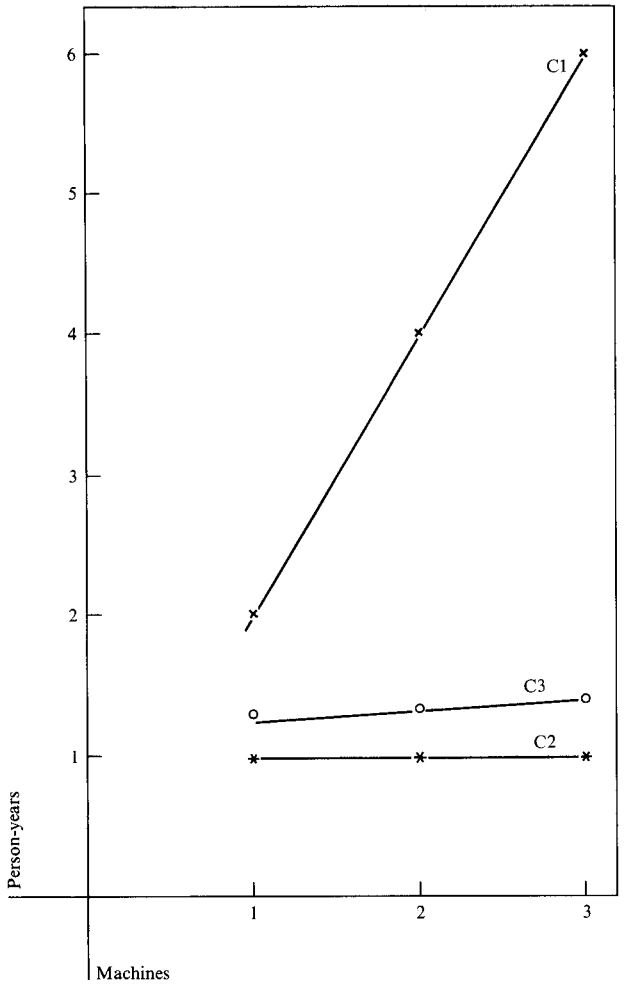

Figure 5 Total programming effort of different approaches vs number of machines, where × indicates assembly language approach, ○ IL approach, and * high level language approach.

1. The first solution would be to build one APL interpreter directly in the assembly language of each machine. Let W_1 be the cost of building each of these APL interpreters. The total cost of this approach would be

$$C_1 = n \times W_1. \tag{1}$$

2. The second approach would be to write a universal APL interpreter in one of the present high level languages. We assume that a compiler for the base language is available for every machine and implements the required subset of the language. Let W_2 be the cost of writing the high-level-language-based APL interpreter. The total cost of this approach would be

$$C_2 = W_2. \tag{2}$$

3. Our approach consists in the use of IL as the system programming language. Let W_3 and W_4 be the cost of writing the IL-based APL interpreter and the cost of building an IL compiler, respectively. The total cost of this approach would be

$$C_3 = W_3 + n\, W_4. \tag{3}$$

We have previously built [2] an APL interpreter in assembly language at the cost of about four person-years. The cost of writing each assembly language APL interpreter, W_1, would be much lower, because of our previous experience and because most algorithms would be available. We estimate it at about two person-years.

We have already written an IL APL interpreter at a cost (W_3) of over one person-year, a smaller figure than the one estimated for W_1, due to the programming and debugging ease provided by the high level syntax of IL. We also assume it is not possible to write an APL interpreter in any high level language at a cost (W_2) lower than one person-year.

A first compiler written in APL and translating IL programs into IBM System/370 assembly language has been built at a cost of two person-months. The cost of changing the code generator so as to translate IL into the assembly languages of IBM Series/1 and another experimental computer was only one person-month. We take this to be the value of W_4.

Substituting the indicated values for W_1, W_2, W_3, and W_4 in Eqs. (1-3), we find (see Fig. 5)

$$C_1 = 24 \times n, \tag{4}$$

$$C_2 = 12, \tag{5}$$

$$C_3 = 14 + n. \tag{6}$$

To gain insight into the relative merits of the three approaches in terms of size, we have written several portions of the APL interpreter (those most frequently executed [9], totaling about 3 percent of the whole program) in IL, FORTRAN, PL/I, and directly in assembly language, and we have compiled them into System/370 machine language. The FORTRAN programs have been compiled by FORTRAN G and also extended FORTRAN H compilers with the option for maximum optimization. The results are shown in Table 1, where the figures refer to the size in bytes of the part of the object program corresponding to executable instructions. The figures give directly a good estimate of size. Since the same algorithms were used in all cases, execution speed may also be roughly estimated as being proportional to the number of machine instructions generated and thus to the space these instructions occupy. In order to get a better estimate, the number of instructions within the inner loops and the number of times they are executed should be taken into consideration.

We may estimate from the above examples that there is a loss of efficiency of about 10 percent when writing the program in IL as compared to assembly language. On the other hand there is a loss of at least 70 percent when the interpreter is written in FORTRAN or 150 percent when it is written in PL/I, respectively (even with the optimizing compilers available), as compared to the program written in IL.

The figures given in Table 1 contain in all instances the overheads associated with call-return management, data management, and so forth, allowing the chosen examples to be incorporated into a full running interpreter, so that they are truly comparable.

We have seen above that approaches 2 and 3 are much more advantageous than approach 1 with respect to programming effort measured in person-time. Now, comparing approaches 2 and 3, we see that the latter, while slightly unfavorable with respect to programming effort (with the assumption that FORTRAN or PL/I compilers will ever be available), from a space point of view is considerably better. We have thus chosen the IL approach, which meets our severest requirement, namely, limiting the size of the interpreter.

The reason for the negligible overhead of IL programs compared with equivalent assembly programs does not lie in optimization properties of the compiler (which would have made it too complex to be written in two months) but in the semantic closeness of IL to assembly language; many of the IL primitive operations produce a single object instruction. The only optimization feature introduced in the compiler design is the propagation of the constants or parameters contained in the registers of the machine in order to save load and store instructions.

Optimization is thus a responsibility of the IL programmer, as is also the case with any assembly programmer. This agrees with our purpose in using IL as a substitute for assembly languages, not for high level languages.

If the need for even better performance arises once the interpreter has been compiled for a given machine, an assembler programmer (who obviously need not know IL) could manually optimize the object assembly program by taking advantage of the special instructions of the machine.

State of the work

An APL-IL interpreter has already been written, in which the full APL language and a set of system commands have been implemented. It includes an editor to build and modify user functions.

The first IL compiler we built translates IL programs into System/370 assembly language, in order to profit from the fact that one of these machines is available to us. This compiler has allowed us to translate and test the interpreter, which is now being debugged. Two other compilers are already available, translating IL into the assembly languages of the Series/1 and an experimental computer.

Table 1 Results using different approaches.

Program	PL/I	FORTRAN *compiled with*		IL	*Assembler*
		F-G	F-X		
Lexical analysis of numeric constants	2690	2324	1924	1260	1096
Syntax analysis of constants and variables. Assignment function	2504	2062	1368	702	640
Vector catenation	3302	2762	1900	1044	964
Total	8496	7148	5192	3006	2700

A machine-dependent supervisor has been added to the System/370-translated interpreter to provide management of the work space library and terminal input/output, resulting in a prototype system that is currently being used to test the interpreter and to compare its performance with that of APLSV, also available in the same machine. The translated interpreter occupies a total space of 74 Kbytes (where K = 1024), which is less than that needed for the APLSV interpreter. Execution speed is not easily compared, because different algorithms have been used in both systems (ours trying to minimize size). However, figures currently obtained indicate that our system is, on the average, about 1.15 times slower than APLSV.

Conclusions

The systems programming approach has been found optimal to solve the problem of building a machine-independent APL interpreter. However, a special systems programming language has been designed to meet all the requirements of our problem. The intermediate level language IL has a higher level syntax than assembly languages. Its semantics are closely related to those of assembly languages, notwithstanding the fact that it maintains machine independence. This approach eases programming, debugging, and readability (because of its syntax) compared to assembly languages. Also, compilers for IL, producing efficient code, can be built at little cost (because of its semantics).

We are using IL as a tool for systems programming, in order to build a universal APL interpreter. It could also serve as a kind of machine-independent assembly language, once compilers for different machines have been built.

We do not intend to present IL as an alternative to programming in high level or other systems programming languages. It was designed to meet the severe requirements imposed by our particular application, i.e., a universal APL interpreter implementable both in small and large computers.

Languages similar to IL could be used in computer science education as substitutes for assembly languages. IL is easy to learn. Two staff members at this Scientific Center who are fluent in FORTRAN, PL/I, and APL, but who had never written programs in assembly language, were able to write and successfully execute their first IL program one day after they had been given the manual for the language. The language is completely designed, and compilers for three different machines have been built. The universal APL interpreter has been completed and is in the process of being debugged.

Appendix A: Formal syntax of IL

In the following representation, λ represents the null string.

```
<PROGRAM> ::= <DECLARATIONS> <SECOND PART>
<SECOND PART> ::= λ | <SEPARATION> <EXECUTABLE STATEMENTS>
<SEPARATION> ::= / <END OF STATEMENT> | //<END OF STATEMENT>
<DECLARATIONS> ::= λ | <DECLARATION STATEMENT><END OF STATEMENT>
                                                    <DECLARATIONS>
<DECLARATION STATEMENT> ::= <I.V. ASSIGNMENT> | <EQUIVALENCE>
<I.V.ASSIGNMENT> ::= VARIABLE NAME←<VNL><DIMENSION><VALUES> |
                          LABEL NAME ← <DIMENSION> LABEL NAME <LIST3>
<VNL> ::= λ | VARIABLE NAME ← <VNL>
<DIMENSION> ::= λ | INTEGER CONSTANT ρ
<VALUES> ::= CONSTANT <LIST1> | PARAMETER <LIST2>
<LIST1> ::= λ | BLANK CONSTANT <LIST1> | , <VALUES>
<LIST2> ::= ⧈ λ | , <VALUES>
<LIST3> ::= λ | , LABEL NAME <LIST3>
<EQUIVALENCE> ::= VARIABLE NAME <INDEX1> = <EQU OBJECT> |
                                        PARAMETER = <PARAMETER EXPRESSION>
<EQU OBJECT> ::= VARIABLE NAME <INDEX2> | POINTER
                                                  [<PARAMETER EXPRESSION>]
<INDEX1> ::= λ | [<OBJECT1>]
<INDEX2> ::= λ | [<PARAMETER EXPRESSION>]
<OBJECT1> ::= VARIABLE NAME | <PARAMETER EXPRESSION>
<PARAMETER EXPRESSION> ::= <CONSTANT OBJECT> | <PARAMETER EXPRESSION>
                                    <DYADIC FUNCTION> <CONSTANT OBJECT>
<EXECUTABLE STATEMENTS> ::= λ | <LABELED STATEMENT> <END OF STATEMENT>
                                                <EXECUTABLE STATEMENTS>
<LABELED STATEMENT> ::= <LABEL><STATEMENT> | LABEL NAME :
<LABEL> ::= λ | LABEL NAME :
<STATEMENT> ::= <ASSIGNMENT> | <TRANSFER> | <ROUTINE CALL> |
                                                        <RETURN> | SAVE
<ASSIGNMENT> ::= POINTER → <INDEXED VARIABLE> | POINTER Δ <EXPRESSION> |
                                    <INDEXED VARIABLE> ← <EXPRESSION>
<INDEXED VARIABLE> ::= VARIABLE NAME <INDEX>
<INDEX> ::= λ | [<PARAMETER EXPRESSION>]
<EXPRESSION> ::= <OBJECT><DYADIC FUNCTION><EXPRESSION> |
                          <MONADIC FUNCTION><EXPRESSION> | <OBJECT>
<OBJECT> ::= ⧈ <INDEXED VARIABLE> | CONSTANT | PARAMETER
<DYADIC FUNCTION> ::= ← | + | - | × | ÷ | ↑ | ↓ | ∧ | ∨ | ⊛ | Δ | | | →
<MONADIC FUNCTION> ::= - | | | ~
<TRANSFER> ::= → LABEL NAME <CONDITIONAL STATEMENT>
```

```
<CONDITIONAL STATEMENT> : : = λ | IF <CONDITION>
<CONDITION> : : = <INDEXED VARIABLE> <RELATION> <EXPRESSION> |
                                  <INDEXED VARIABLE> ∧ <CONSTANT EXPRESSION> |
                                                        <CONSTANT EXPRESSION>
<CONSTANT EXPRESSION> : : = <CONSTANT OBJECT> <DYADIC FUNCTION>
                                  <CONSTANT EXPRESSION> | <MONADIC FUNCTION>
                                   <CONSTANT EXPRESSION> | <CONSTANT OBJECT>
<CONSTANT OBJECT> : : = INTEGER CONSTANT | PARAMETER
<RELATION> : : = < | ≤ | = | ≠ | > | ≥
<ROUTINE CALL> : : = EXTERNAL ROUTINE NAME | LABEL NAME
<RETURN> : : = RET | → INTEGER CONSTANT <CONDITIONAL STATEMENT>
<END OF STATEMENT> : : = <COMMENT> CARRIAGE RETURN
<COMMENT> : : = λ | ⍝ <CHARACTER STRING>
<CHARACTER STRING> : : = λ | NON CARRIAGE RETURN CHARACTER
                                                        <CHARACTER STRING>
```

Appendix B: Identifier type conventions

Type of identifier	*First letter of identifier*
Integer	
One memory unit per element	O R T U V W
Two memory units per element	I J K L M N
Four memory units per element	A B C D G H
Rational	F
Pointer	P
Label	E
Routine names	S
Parameters	X Y Z

References and notes

1. *APL Language*, Publication No. GC26-3847-0, IBM Corporation, Data Processing Division, White Plains, NY, 1975.
2. M. Alfonseca, M. L. Tavera, and R. Casajuana, "An APL Interpreter and System for a Small Computer," *IBM Syst. J.* **16,** 18 (1977).
3. *Series/1 Model 5-4955 Processor and Processor Features Description*, Publication No. GA34-0021, IBM Corporation, General Systems Division, Atlanta, GA, 1977.
4. *VS APL Program Logic*, Publication No. LY20-8032-1, IBM Corporation, Data Processing Division, White Plains, NY, 1976.
5. VM/370-CMS and VSPC.
6. IBM 1620, 1130, 7040, 7090, S/3, S/7, S/360, S/370, Series/1.
7. M. Alfonseca, "An APL-written APL-subset to System/7-MSP Translator," *Proceedings of the APL Congress 73*, North-Holland, Amsterdam, 1973.
8. R. Aguilar, M. Alfonseca, and J. Bondia, *APL to System/7 Assembler Compiler*, SCR.05.73, Centro de Investigacion UAM-IBM, Madrid, 1973.
9. H. J. Saal and Z. Weiss, "Some Properties of APL Programs," *Proceedings of the APL Congress 75*, ACM, New York, 1975.
10. *Machine Oriented Higher Level Languages*, W. L. van der Poel and L. A. Maarssen, eds., North Holland Publishing Co., Amsterdam, 1974.

Received September 12, 1977; revised January 10, 1978

The authors are located at the IBM Scientific Centre, Madrid, Spain.