The APL IL Interpreter Generator

by M. Alfonseca
D. Selby
R. Wilks

The objective of the APL IL Interpreter Generator is to solve the problem of creating APL interpreters for different machines at a minimum cost. The objective has been accomplished by writing an APL interpreter in a specially designed programming language (IL) that has very low semantics but high-level syntax. The interpreter is translated to each target machine language by easily built compilers that produce high-performance code. The paper describes IL, the APL interpreters written in IL, and the final systems generated for seven different target machines and operating systems. Some of these systems have been generated in an extremely short time.

Among the many languages used to write programs, APL and its successor, APL2, are very powerful. They support highly structured data of several different internal types and recognize a large number of primitive functions and operators, some of which (for example, **execute**, ⍎) are extremely complicated for some arguments. The existence of these primitives makes it very difficult for APL to be compiled (except for subsets of the language or through the inclusion of an interpreter in the machine code). Thus full APL and APL2 systems have to be interpretive. These interpreters are very large programs, consisting of tens of thousands of instructions.

Since interpreted programs normally run at least an order of magnitude slower than their compiled equivalents, programs written in APL or APL2 start with a speed handicap as compared to programs written in, say, C. However, the designers of APL and APL2 and the implementers of the interpreters have tried to reduce this effect in two different ways:

- By extending the language with ever more powerful primitives. In a single stroke, these perform complex operations that, in other languages, would require complicated algorithms. In this way, the time for interpretation is minimized with respect to the time for execution. The fact that most APL primitives apply to entire arrays also helps in this direction.
- By programming the interpreters in very low-level languages that make the best possible use of the resources of the machine or the operating system.

As a result, APL and APL2 interpreters were usually written in assembly languages, with the consequent loss of portability. It has been estimated several times that, done in this way, the full development of an APL system for a new machine requires a total of about 30 person-years.

The APL IL Interpreter Generator started as a project in the IBM Madrid Scientific Center in 1977. The objective of this project was to solve the problem of obtaining APL interpreters for different machines, at a minimum cost. The solution was to write an APL interpreter in a programming language, specially

designed for the purpose, that has very low semantics but high-level syntax. This interpreter is translated to each target machine by appropriate, easily built compilers that produce high-performance code.

In the past 14 years, the programming language called the Madrid Scientific Center Intermediate Language (IL) has reached its third version; it has been essentially stable since 1980. The first section of the paper describes the language design decisions, which in many cases are curiously parallel to those made in the design of the C language, although there are important differences. The second section of the paper describes the different interpreters that have been written in IL since 1980. Finally, the last section describes the procedure used to generate an APL system for a given target environment (a machine and an operating system).

The Intermediate Language

The Madrid Scientific Center Intermediate Language (IL) was designed in the late 1970s, according to the following criteria: on the one hand, a high-level syntax was desirable to assure portability between different machines and operating systems; on the other hand, very low-level semantics would make it possible to obtain highly optimized code with very simple, easy-to-build compilers.

The procedure that was followed to design the IL instructions was to select the most common operations in the assembly languages of different IBM machines and to represent them with a high-level syntax. In this way, compilation of IL instructions into assembly language usually becomes a one-to-one translation between one IL symbol and one assembly instruction.

Even control instructions were subject to this procedure. Since the only control instruction in assembly languages is usually the *branch on condition*, this instruction is the only one that was implemented in IL, although it received a high-level syntax in the following way:

```
→label IF condition
```

Optimization, in this kind of intermediate language, is not a question to be solved by the compilers, which we want to build as quickly as possible, but by the IL programmers who write the APL interpreter. Remember that this job should be done only once, although there may be as many compilers as there are different target machines.

The only assumption about the machine in which IL may eventually be implemented is that its memory is considered to be a vector of fixed but undefined size (eight bits or more per byte; two, four, or eight bytes per word). Memory units should be consecutively numbered.

The four elements of IL are now described.

Constants. Constants can be numeric or literal. In actual fact, a literal constant can also be considered as numeric and operated on accordingly. This means that an expression such as

```
'A'+1
```

is valid and (assuming ordinality in the character set) is equivalent to constant

```
'B'
```

The C language manages character constants in the same way.

ASCII (American Standard Code for Information Interchange) or EBCDIC (extended binary-coded decimal interchange code) can be selected as the internal representation of the literal constants. In the case of the APL IL interpreters, ASCII has been chosen.

Numeric constants can be either integer or floating point. Floating-point constants, such as `2.0`, are distinguished by the presence of the period from integer constants, such as `2`.

Identifiers. Identifiers are names that begin with a letter other than Q (which is reserved) and continue with any (possibly empty) combination of letters and figures. The maximum number of characters in an identifier is five.

What an identifier represents is controlled by its first letter, according to Table 1.

A full-word variable has an implementation-dependent length. Depending on the machine (in a 16-bit system, for instance), a full-word variable can be the same as a two-byte variable. This type is, to a certain extent, similar to the **int** type in the C language, but IL does not distinguish full-word in-

Table 1 Identifiers and their definitions

Identifier	Representation
O,R,T,U,V,W	A variable whose value is a vector of one-byte integers or literals.
I,J,K,L,M,N	A variable whose value is a vector of two-byte integers.
A,B,C,D,G,H,P	A variable whose value is a vector of full-word integers or pointers.
F	A variable whose value is a vector of floating-point values.
E	An internal label in a program.
S	A public label in a program.
X,Y,Z	A named constant.

tegers from pointers. Assembly languages do not usually make this distinction either.

The only data structure supported is the vector (a succession of values at consecutive locations). Higher structures (such as matrices) are not a part of IL, as they are not a part of assembly languages. A scalar is considered to be the same as a vector of one element.

Declarations. In an IL program, declaration instructions are located at the beginning and clearly separated from executable instructions. Every variable used by a program must be declared, either by assigning initial values to it, or by defining an equivalence.

Initial values are assigned by means of instructions such as the following:

```
A←1 3 5 7
B←10ρ0
W←'ABC'
```

The first instruction defines A as a vector of four full words with initial values of one, three, five, and seven. The second defines B as a vector of ten full words with initial values of zero. The third defines W as a vector of three bytes with initial values equal to the ASCII representation of letters A, B, and C.

Equivalences are very powerful and have different forms, such as:

```
C=A[2]
V=8ρF
C1=3ρP1(4)
```

The first instruction defines variable C to have the same address as the third element of vector A (zero origin is used). Both A and C are full-word objects by virtue of their initial letter.

The second instruction defines V as a vector of eight bytes, sharing the address of floating-point variable F. This means that V is the vector of the bytes that make up the floating-point value of F, assuming that floating-point values are represented in eight bytes.

The third instruction defines $C1$ as a vector of three full words whose address is the current value of pointer $P1$ plus four. Of course, if the value of $P1$ changes, the address of C will change accordingly.

Pointers are extremely useful in IL programs, just as they are in C. However, there is no restriction on the number of equivalences that may be defined to a pointer at the same time. For instance, the following declarations

```
A1=4ρP(0)
I1=P(0)
V1=P(0)
```

are all valid and define three variables that share the same address (the value of pointer P), but have a different type. $A1$ is a pointer or full-word integer vector of four elements. $I1$ is a two-byte scalar, and $V1$ is a one-byte scalar.

Executable instructions. Executable IL statements are analyzed and executed from right to left. Functions are executed without any precedence rules in the order in which they are found. Parentheses are not allowed. The main IL executable instructions are of two different types: assignment instructions and execution control instructions.

Assignment instructions may take four different forms, according to the following syntax:

```
variable ← expression
variable Δ expression
variable ∇ expression
pointer_variable → address expression
```

where the first form corresponds to normal assignment, the second increments the value of the variable by the right-hand expression, the third decrements that value in the same way, and the fourth, only applicable to pointers, assigns to the variable the address of the expression on the right side.

Execution control statements have three different forms:

```
→label
→label IF condition
→label_list OF index
```

where the first form corresponds to the unconditional transfer, the second to the conditional transfer, and the third to a computed go-to instruction.

The operations that can be a part of an expression are the typical ones usually encountered in most machine languages, such as the following: addition (+), subtraction (−), multiplication (×), division (÷), residue (\|), bit shift to the left (↑), bit shift to the right (↓), bit-to-bit logical operations that include not (~), and (∧), inclusive or (∨), and exclusive or (∘), absolute value (\| in monadic form), and an operation to compute the integer part of a floating-point number (~ in monadic form).

The following is an example of an executable instruction in IL:

```
P1←AREF+ZEI4↑4+1↑DREFI
```

This instruction computes the value of pointer P1 in the following way: The value of variable *DREFI* is multiplied by two (a shift to the left of one position is equivalent to a multiplication by two); then, four is added to the preceding result. Next, the new result is shifted to the left by as many positions as the value of constant ZEI4 (which depends on the target machine). Then the value of variable *AREF* is added, and finally, the result is assigned to pointer P1.

Another kind of executable instruction is the subroutine call. Its syntax is very simple, just the name of the subroutine. No parameters can be passed explicitly. All of them must be passed through common memory, or by means of a set of special pointer variables, the values of which are automatically restored before returning to the calling routine. These variables fulfill the role of the machine registers, and in fact, in several of our implementations they are registers, but this is not necessarily so.

Language tradeoffs. A question that could be discussed is whether IL has any advantages over C for the implementation of machine-independent software. This question is really after the fact since IL was designed in 1977, at a time when C was in its infancy and far from being as widespread as it is now. However, in our opinion, IL is superior to C in its memory management capabilities, which are much nearer to the machine language level, and also in its ability to define multiple pointer-based structures that can overlap freely and move around without any restrictions.

In contrast, C has better type-constraint capabilities that provide the programmer with mechanisms to detect certain errors at compile time, which IL compilers do not have. However, we did not find the lack of these capabilities frustrating in our development of APL interpreters.

Finally, IL, being a less complicated language, can be translated by very simple compilers. This point was important in our development procedure, which is described in the last section of this paper.

The APL IL interpreters

IL has been used for the development of several different interpreters. In the time from 1978 to 1982, an APL interpreter was built at about the same level of the language as the one implemented in the VS APL product. This interpreter was especially applicable to small machines with reduced data spaces in memory, and to increase the amount of workspace available to the user, we introduced the concept of an elastic workspace. This interpreter was compiled into the System/370* (which we used as our test machine), the Series/1*, and the IBM Personal Computer.

The Series/1 computers had an important limitation: the memory data space used by one application was restricted to 64K bytes. To increase it, we implemented the elastic workspace as a disk extension of the workspace. APL objects directly accessible to the user (in the active workspace) could also reside on disk and would be copied into the main memory only when they were needed.

When the IBM Personal Computer was announced in 1981, we decided to translate our interpreter to

this machine under the Personal Computer Disk Operating System (PC DOS, or DOS). In this case, there was the same limitation in the fact that the use of segment registers made only 64K bytes directly available. But these machines have greater flexibility in comparison to the Series/1, since the contents of the segment registers can be changed by the program. This flexibility made it possible for us to implement the elastic workspace extension in main memory, which made it much faster and more efficient.

The workspace was divided into two different sections. In the first section, with a length of 64K bytes, all the objects were directly accessible to the programs. This section included the APL symbol table, the APL execution stack, and many APL objects, all of them smaller than 32K bytes.

The second section (the elastic workspace) contained APL objects larger than 32K bytes and (possibly) smaller APL objects that did not fit in the directly available workspace at a given time and were not currently needed. Depending on the amount of space available (limited in DOS to 640K bytes but possibly reduced by the actual memory of the machine and the loading of the operating system extensions), the elastic workspace could be automatically reduced to zero.

This organization made it possible to build the IL compiler for the IBM Personal Computer in such a way that the compiler could assume that all of the objects are directly accessible and forget about segment registers. The only module not complying with this restriction was the handler of the elastic workspace, which was written directly in assembly language.

However, the indicated memory organization had an important disadvantage: many of the basic APL structures, such as the symbol table and the execution stack, could not increase further than 32K bytes, and users soon found that this was a strict limitation. Therefore, during 1983–85, we developed a new APL interpreter with a more general workspace management, specially adapted for 16-bit addressed segmented microprocessors (such as the i8086). This interpreter, which from the language point of view was still at the VS APL level, was compiled into the System/370 (which we always use as the test machine) and also into the IBM Personal Computer (under DOS) and the IBM Japanese Personal Computer and JX PC (under Japanese DOS) as a result of a joint project between the IBM Madrid and Tokyo Scientific Centers.

The elastic workspace concept was abandoned, or (as it may be preferred) extended to the whole workspace. In actual fact, what happened is that this system incorporated a single workspace area containing all of the APL objects, including the symbol table and the execution stack, regardless of their sizes. The lower part of the workspace, however, always directly accessible through the base segment registers, includes all of the interpreter data and work areas plus four "operand areas."

This organization made it possible to build the IL compiler for the IBM Personal Computer so that the compiler could assume all of the objects are directly accessible.

An operand area is a section of the workspace located in the lower 64K bytes of the total workspace, where the system can copy APL objects, either completely or partially. A set of special subroutines manages the transfer of the data from the operand areas to the workspace proper and vice versa. The remainder of the interpreter works only with the operand areas and can thus forget about the segment registers. Only a few modules in the whole interpreter (less than 10 percent) must work directly on the workspace, and thus they must be hand-modified in assembly language to introduce the required modifications to the segment registers.

In 1985 we started a joint project between the Madrid and United Kingdom Scientific Centers to build an APL2 interpreter written in IL. This interpreter has been compiled, as usual, into the System/370, and also to the following target machines and operating systems: the IBM Personal Computer (IBM PC), Personal Computer AT*, and Personal System/2* (under DOS and Operating System/2*, or OS/2*), the IBM Japanese Personal Computer (under Japanese DOS), the Intel 80386**-based machines in 32-bit addressing mode (under DOS), the

Table 2 Previously available interpreters

1. IBM Personal Computer APL, version 1.0, Program Number 6024077, 1983.
2. IBM Personal Computer APL, version 2.0, Program Number 6391329, 1985.
3. 5550 NiHonGo (Japanese) APL, version 1.0, Program Number 5600-JPL, 1984, developed in collaboration with the Tokyo Scientific Center.
4. 5550 NiHonGo (Japanese) APL, version 2.0, Program Number 5600-JPN, 1985, developed in collaboration with the Tokyo Scientific Center.
5. JX NiHonGo (Japanese) APL, Program Number 5601-JPL, 1985, developed in collaboration with the Tokyo Scientific Center.
6. APL2 for the IBM Personal Computer, version 1.0, Program Number 5799-PGG (PRPQ RJ0411, Part No. 6242936), 1988.
7. APL2 for the IBM Personal Computer, version 1.0E, Program Number 5604-260 (Part No. 38F1753), and Program Number 5775-RCA (Part No. 38F1754), 1988.
8. APL2 for the IBM RISC System/6000, Program Number 5765-012, 1991, developed in collaboration with the APL2/6000 Development Group from the IBM Kingston Laboratory.

IBM 6150 RT PC* (under Advanced Interactive Executive*, or AIX*), and the IBM RISC System/6000* (under AIX).

There are two versions of this interpreter. The first one, used to generate the PC-like 16-bit systems, still uses the memory management described for the second APL interpreter. However, in the second APL2 interpreter used to generate the 32-bit systems, where memory management is not a problem, some of the modules have been replaced by others that work directly on the workspace, skipping the copy to the operand areas, to improve performance.

An additional improvement in the APL2 interpreters is the presence of a reference table, functionally intermediate between the symbol table and the actual APL objects. This improvement means that most of the time the interpreter may refer to a given object by its reference number, regardless of the actual position where the object is located in the workspace. There are several important consequences of this organization that are now described.

On the one hand, a given APL2 piece of data may be pointed to by more than one APL object. Since APL2 supports general arrays, this capability is important to prevent memory duplication. The reference table keeps information that indicates whether an object is multipointed, which will be used in case of modification to decide whether the value should be copied somewhere else before the changes are performed.

On the other hand, garbage collection is much simplified and made extremely fast. This has always been the case with APL, but it is even more dramatic now that extremely large workspace sizes can be attained. With our 32-bit interpreter, workspace sizes can reach many megabytes, but even so, garbage collection never takes longer than a few seconds. This speed contrasts with other interpretive languages, such as LISP and Smalltalk, where garbage collection was traditionally a very expensive procedure, sometimes taking several minutes to complete.

Table 2 lists some of the previous interpreters that have become international IBM products.

Generating an APL interpreter

The procedure to generate an APL2 system for each environment (machine and operating system) can be summarized as follows. First, a compiler that translates IL code into the target machine code is built. Next, the APL2 IL interpreter is compiled into the target machine code. This produces an incomplete system, with a few loose ends (subroutines) that depend on the operating system and that have not been written in IL. These subroutines are then written, usually in assembly language, and added to the compiled interpreter. Finally, some auxiliary processors are written to perform special I/O operations.

This procedure has proved its usefulness in the fast and effective generation of APL2 interpreters for different machines. The outstanding example was the i80386 interpreter, where we could get rid of the fourth step (since we took care that all auxiliary processors written for the IBM Personal Computer and Personal System/2 [PS/2*] interpreter would be compatible). The total effort required to execute the other three steps and produce and debug a full APL2 system for these machines was 13 person-weeks. The system was announced and shipped just six months after the work started.

Another outstanding example was the porting of the APL2 IL interpreter to the IBM RISC System/6000 machine under the AIX operating system. It was done in about ten person-weeks by two people who did not have previous knowledge of either IL or the RISC System/6000 machine code.

IL compilers. The IL compilers are usually written in APL or APL2, which makes them very easy to adapt to new target machines. They are somewhat slow since they are being interpreted, but this is not a problem since, in principle, they need only be executed once.

When an APL system must be generated, it is usually not necessary to build a full IL compiler.

When an APL system must be generated for a new machine or operating system, it is usually not necessary to build a full IL compiler. Since the source language is the same, the lexical and syntax analysis sections of any of the preceding compilers are automatically usable. Only the code generator section must be rewritten, and even there, many subprograms and program structures can be reused.

The exception is the IL-to-System/370 compiler, since we are using the System/370 as a test machine and many changes and trials are performed on it. Therefore, the IL-to-System/370 compiler was written in IL and is much faster than all of the other IL compilers.

At this point, we have IL compilers available for the System/370, the Series/1, the i8086 and i80286 machines (which include the IBM Personal Computer, the PS/2 Models 25, 30, 50, and 60 and the Japanese IBM PC), the i80386 machines (such as PS/2 Models 70, 80, 90, and 55SX), the IBM 6150, and the IBM RISC System/6000. The last three compilers are written in APL2.

Operating-system-dependent code. Operating-system-dependent code performs those functions that depend closely upon the operating system and are not easily made machine-independent. They include system initiation and disconnection, machine check recovery, console I/O, sequential file I/O, and the timer routines. This code, as compared to the size of the APL interpreter, amounts to about 5 percent of the total code.

In the case of the Series/1, we also implemented a time-sharing system able to support the simultaneous use of the machine by several users. This system was written directly in assembly language, and its presence increased the amount of machine-dependent code to about 10 percent of the total code of the system.

Auxiliary processors. Auxiliary processors are written for the management of different peripherals and specialized computations. They perform functions such as loading and execution of external programs, printer interface, operating system interface, full screen management, data file processing, communications, graphics, music generation, special device drivers, and logic programming.

Not all of these auxiliary processors are available for all of our target machines. Some of them are written in IL, some in C, and some in assembly language. A few of them (such as the special device drivers) are not only machine- and operating-system-dependent, but also hardware-attachment-dependent. It makes no sense to develop them in a high-level language, since assembly language always provides the maximum efficiency.

Conclusion

The APL IL Interpreter Generator has proved its usefulness in generating APL and APL2 interpreters with a considerable reduction of the total product cycle. It has been used to generate nine IBM products: the eight APL and APL2 systems listed previously, plus an educational product announced by IBM Japan, called LETSMATH, that includes the interpreter without the user being aware of it. Several additional systems, restricted for IBM internal use, have also been generated in the same way.

* Trademark or registered trademark of International Business Machines Corporation.

** Trademark or registered trademark of Intel Corporation.

Cited references

1. *APL Language*, GC26-3847, IBM Corporation (1975); available through IBM branch offices.
2. *APL2 Programming: Language Reference*, SH20-9227, IBM Corporation (1987); available through IBM branch offices.
3. M. Alfonseca and M. L. Tavera, "A Machine-Independent APL Interpreter," *IBM Journal of Research and Development* **22**, No. 4, 413–421 (July 1978).
4. M. L. Tavera, M. Alfonseca, and J. Rojas, "An APL System for the IBM Personal Computer," *IBM Systems Journal* **24**, No. 1, 61–70 (1985).
5. M. Alfonseca and D. Selby, "APL2 and PS/2: The Language, the Systems, the Peripherals," *APL89 Conference Proceedings, APL Quote Quad* **19**, No. 4, 1–5, ACM, New York (1989).

Accepted for publication June 21, 1991.

Manuel Alfonseca *IBM Software Technology Laboratory, Paseo de la Castellana, 4, 28046 Madrid, Spain.*

Dr. Alfonseca is a Senior Technical Staff Member in the IBM Software Technology Laboratory. He has worked in IBM since 1972, having been previously a member of the IBM Madrid Scientific Center. He has participated in a number of projects related to the development of APL interpreters, continuous simulation, artificial intelligence, and object-oriented programming. Eleven international IBM products have been announced as a result of his work. Dr. Alfonseca received electronics engineering and Ph.D. degrees from Madrid Polytechnical University in 1970 and 1971, and the Computer Science Licenciatura in 1972. He is a professor in the Faculty of Computer Science in Madrid. He is the author of several books and was given the National Graduation Award in 1971 and two IBM Outstanding Technical Achievement Awards in 1983 and 1985. He has also been awarded as a writer of children's and juvenile literature.

David Selby *IBM United Kingdom Scientific Centre, Athelstan House, St. Clement Street, Winchester, Hants, SO23 9DR, England.*

Mr. Selby joined IBM at the Havant manufacturing location in 1977, where he worked on many APL projects in the capacity of analyst programmer and later as a microcode engineer for the 4700 Finance Industry System. In 1985 he joined the Scientific Center at Winchester in the Graphics Systems Research group as a scientist employed on workstations. Beginning in 1983, he collaborated with Dr. Alfonseca on APL/PC 2.0 with special emphasis on auxiliary processors and device support. The result of this work was used in APL/PC 2.0, Japanese PC APL 2.0, and the APL2/PC products. Mr. Selby is also responsible for the design of the extended memory driver of the 32-bit version of the APL2/PC interpreter and has worked as a technical consultant to the APL2/6000 project. While working for IBM, he has obtained an ONC in electrical engineering, and an HNC in computer science. He has also received an IBM Exceptional Achievement Award.

Ron Wilks *IBM United Kingdom Scientific Centre, Athelstan House, St. Clement Street, Winchester, Hants, SO23 9DR, England.*

Mr. Wilks joined IBM in 1973 and has used APL since his very first day with IBM. He started his career in a group developing diagnostics for small disk files. From there, he joined the IBM Hursley Information Systems (IS) Applications Support group to support APL and related products. While in IS, Mr. Wilks assisted with enhancements to the APL/PC Version 1 product culminating with the announcement of the APL/PC Version 2.0 product for which he received an IBM Exceptional Achievement Award. After IS, he joined the small group of APL2/PC developers to assist with APL2 for the IBM PC product and, more recently, the AIX APL2/6000 product.

Reprint Order No. G321-5448.