# COMPILING APL

**Robert Bernecky***
Snake Island Research, Inc.
18 Fifth Street, Ward's Island
Toronto, Canada M5J 2B9
Phone: (416) 203 0854
email: bernecky@acm.org

May 1990

## Abstract

A study into the creation of efficient, portable applications for a variety of supercomputer architectures resulted in a prototype APL compiler, called ACORN, targeted at the Cray X-MP [1] and the Thinking Machines Connection Machine CM2 [2]. ACORN used a number of advanced features of APL, including recursive data structures (boxed arrays) and the rank adverb, which considerably simplified the effort required to produce the compiler. The use of APL allowed the compiler to be complete to the point of producing operational code within a week, yet allowed considerable flexibility in altering it to meet changing needs, as performance concerns came into play. Tokenization was performed in parallel, which may shed light on improving the performance of production compilers on parallel architectures.

## 1 Introduction

In 1987, discussions between the author and two scientists [3] at the Dallas Research Laboratory of Mobil Research and Development Corporation resulted in a joint project to study *the use of APL as a delivery vehicle for parallel computation.* The project was intended as no more than a feasibility study, to determine if APL could, in fact, compete with FORTRAN in the supercomputer arena.

The original project envisioned hand-compilation of two relatively simple seismic applications – convolution and Normal Move Out – written in APL, and a comparison of their performance to the production versions of the same applications, written in Cray FORTRAN. Although the applications were less than a hundred lines of APL, the tediousness of hand-compilation led to a decision to write a primitive APL to C compiler, in APL. This decision turned out to be crucial – a working compiler was operational within a week. To be fair, the majority of coding actually resided in the run-time library (the compiler does no optimization nor data flow analysis whatsoever), but the time does show that compiling APL need not be a task of monumental complexity. The remainder of this paper deals with some of the decisions made as part of the compiler design, discoveries made about performance and language design, and discusses some of the internals of the compiler.

*This paper originally appeared in *Arrays, Functional Languages, and Parallel Systems*, Ed. L.M.R. Mullin, *et.al.*, Kluwer Academic, 1991

[1]X-MP is a trademark of Cray Research, Inc.

[2]Connection Machine is a trademark of Thinking Machines Corporation.

[3]Stephen Jaffe and George Moeckel.

# 2 Which APL should be compiled?

A sensible choice of an APL dialect to compile would be ISO Standard APL [IS84]. Although this is a standard, its use was beyond the scope of our resource-limited project, and the language, like any other language which has evolved over time, contains a number of warts in its design. We initially decided to compile a subset of ISO Standard APL, with a number of simplifying restrictions.

## 2.1 System Variables

`Ìct`, the comparison tolerance used in relational verbs to determine equality, was considered to be always zero. This reduced the effort required to code the run-time library. Like many other restrictions to be discussed, it could be easily lifted in a production compiler. Other system variables were treated similarly, being given a default value which was not alterable or accessible by the application.

## 2.2 Interpreted Verbs

In order to avoid the need to include an entire APL interpreter as part of the compiled code, facilities which require knowledge of the APL symbol table, such as `â` and `Ìfx`, were forbidden.

## 2.3 Data Type Restrictions

Because we were primarily interested in the performance of numerically intensive applications, we took the liberty of making all arrays floating point. Although we expected a performance penalty in areas such as indexing and Boolean algebra, we felt that the ability to avoid all data flow analysis was a reasonable compromise for a feasibility study. If we were unable to compete in large numeric computations, we'd not have lost time writing code which would prove nugatory.

## 2.4 Scoping Rules

APL, like some dialects of LISP, traditionally has used dynamic scoping, in which a called verb can access or alter any variables which belong to its caller, and which are not localized within the called verb. This facility is a mixed blessing, and is more often cursed than praised by application writers who have forgotten to localize a variable, or to supply a label for a branch target. In addition, well-written APL rarely needs such crutches, particularly when recursive data structures are available for use. With this in mind, we chose to adopt static scoping rules, in which all arrays are either strictly local (to the verb which localizes them) or global to the entire execution environment. This decision had two implications.

Firstly, it meant that any APL verb could be compiled separately, with no knowledge of other verbs, calling tree, and so forth. This meant that the compiler does not have to perform global data flow analysis, and that the "compilation set" required was a single APL verb.

Secondly, it meant that the run-time environment need not contain a symbol table to allow it to be aware of the names of APL objects, altering their meaning, depending on the calling tree being invoked. An alternate approach, in which multiple copies of generated code were produced, based on all possible calling trees, was also not required.

The decisions to stick with a single data type, and to adopt static scoping were the keys which made it possible to have the compiler emitting code within a week.

# 3 Declarations and Verb Valence

An early decision was the attempt to avoid declarations or other modifications to APL programs. This, combined with independent compilation of each verb, implied certain things about the interpretation of names. For example, if a name was not localized within a verb, then it could be either another verb or a global variable. Our rather arbitrary decision was to treat such names in the following manner: Niladic verbs were forbidden, except for the "main" verb required by the C language. When an expression such as `abc` was encountered, `c` was assumed to be a variable, because of the ban on niladic verbs. `b` was therefore either a monadic or dyadic verb. If its valence could not be determined by other information (such as `a` being a local variable), it was assumed to be dyadic. If `a` and `b` were in fact both monadic verbs, this fact would not be discovered by the compiler. Presumably, running "lint" on the compiled set of verbs would pick up this error. The user would then have to establish a workaround, through use of an identity such as `aÛbc`, parentheses, as in `a(b c)`, or introduction of a temp, as in `t1ûbcþat1`. The desirability of, and treatment of, declarations remains an open issue for APL compilation, but it is probably more of a theological issue than a practical one.

## 4 Compiler Structure and Internals

The compiler is a model of simplicity. In fact, it is hard to believe just how simple and primitive the compiler is. It is a total of 726 lines of straightforward APL, comprising 111 verbs, in which no attempt was made to optimize performance. About half of the verbs are action verbs invoked by a finite state machine, which will be discussed in more detail later. The code contains 19 branches, of which all but two could be replaced by case constructs, such as those discussed in another paper [Be84]. The remaining two branches are used to loop over tokens and lines of source code. The latter could be replaced by a suitable invocation of the rank adverb, leaving only the finite state machine token loop.

The tokenizer examines the entire source verb in parallel, generating a token table of the same shape as the source verb. The resulting token table and source verb are then used jointly by the remainder of the compiler.

Each tokenized line is compiled independently, with a finite state machine (FSM) invoking action verbs as appropriate. The verb header is treated separately, because of its anomalous syntax. Independent compilation was done for reasons of simplicity and *workspace full* avoidance on the microcomputer. I contemplated a fully parallel compiler, but the storage limitations of the PC precluded any serious work in this area. The main problem was in the area of excessively large stacks when compiling verbs with many levels of parentheses, admittedly a rare case in APL. Irregularities in traditional APL syntax were another annoyance. The FSM (a standard APL reduction parser) is based on the current state of syntax analysis, and the class of the next token(s) to the left.

The compiler makes extensive use of boxed arrays (recursive data structures) and the rank adverb. Boxes are used to represent the symbol table as a matrix of symbol name, class, and value. The expression stack is also represented as a table, with each row containing the syntax analyzer state, and the stacked token or expression at that point. The generated code is kept as a boxed array, sometimes to several levels of nesting, for convenience. A single application of the format verb (thorn) at end of compilation produces the final character matrix result from the previously generated boxed code.

Code generation is facilitated by the use of the SHARP APL link verb, in conjunction with the rank adverb, to permit operations such as gluing fixed prefixes onto a set of declarations. The use of a verb to denote link allows application of adverbs such as rank to modify the action of the link. A typical use within the compiler is the expression `'staticVAR 'Øê1consØê1initers`, to generate declarations and initial values for constants which appear in the verb being compiled.

## 5 Portability

Our initial proposal was to compile to FORTRAN or Cray Assembler Language, CAL. A quick look at the CAL manual convinced me that life was too short to learn to write efficient code in CAL, so I sadly began to reread FORTRAN manuals after a twenty year hiatus. About this time, I discovered that the Portable C Compiler (PCC) had been ported to the Cray X-MP, so the decision was quickly made to use C as our target language. Although PCC did not perform to the same standards as the FORTRAN compiler – it did only the crudest vectorizations – its functional nature made it a much more suitable language for APL compilation. For example, the passing of data structures as arguments and results was trivial in C.

The use of C offered an added advantage in portability – we were able to perform most of our development work on other platforms, including IBM-compatible microcomputers and a SUN 386i workstation. Furthermore, the compiler was able to produce code for a variety of other machines, including mainframes and massively parallel supercomputers.

## 6 Application Portability

A major advantage of the use of APL as an application language is its portability. Because APL is an abstract language, optimizations which are required at the source level in languages such as FORTRAN can be performed by the compiler or the run-time library. APL's rich set of powerful primitives allows commonly required verbs such as set membership, sorting, searching, and matrix manipulations to be tuned appropriately to the underlying hardware by the compiler writer, rather than requiring the application programmer to change the application.

One example of this perfect portability is searching. In a traditional language such as FORTRAN, a programmer would have to specify the search technique explicitly, perhaps as a linear or binary search. In APL, the decision as to the best technique is left to the implementor of the run-time library – perhaps a binary search on a von Neumann machine, and a brute force unit time algorithm on a Connection Machine. In our work, we made extensive changes to the run-time library, but the APL application source code remained, with one exception to be noted below, unchanged.

## 7 I/O Problems

Input and output presented a major design problem for the project. PCC 4.0 did not support any record-based I/O capabilities, which we took to mean that PCC could not read the seismic files written by FORTRAN, so character-based files were used. We later learned that impedance matching routines could be written to allow FORTRAN I/O facilities to be intermixed with those native to C, but we did not have the luxury of time to make use of them.

Because APL systems tend to have component-based I/O systems, which encourage the reading and writing of arrays, rather than records, we created several utility verbs to provide these services. Dealing with files on the Cray was a problem – STDIN and STDOUT forms of I/O are acceptable, but we had no pleasant method of talking about files other than these. Lack of a character data type didn't help, either. We are going to have to come to grips with the issues of windowed I/O, data base access to the external world, and related problems.

The C language also created problems, in that the only real control over application customization lies in the execution shell's command line, a very low capacity interface. This placed C at odds with APL's traditionally functional approach to "main" programs. This was not a severe problem, yet it remains an annoyance.

## 8 Compiler Performance

The performance of the compiler on a mainframe is quite adequate, particularly since no thought was given at all to performance issues. Typical verbs compile in a few tenths of a second on a 3090-class machine.

On SHARP APL/PC, compile times are too slow for any serious applications – about one line per minute on an old PC/XT. This is due largely to the approach that was taken in implementing SHARP APL/PC. The interpreter is IBM/370 machine code, with a 370 emulator interpreting each 370 instruction. As a historical note, this approach was taken because it got a highly reliable and functional APL system (Six years after its introduction, SHARP APL/PC [4]

[4]SHARP APL/PC is available as shareware through various bulletin boards, and from Iverson Software Inc, 33 Major St., Toronto, Ontario M5S 2K9, Canada.

remains the only APL interpreter on a PC which has boxed arrays and complex arithmetic) to the market quickly, and because we thought that IBM would push the XT/370 technology much harder than they actually did. Without an XT/370 card in the PC, the compiler is painfully slow. On a fast AT-class machine, performance is barely acceptable. Compiling the compiler would, of course, be the ideal solution, but that is a task for the future.

Profiling the performance of the compiler showed that the majority of CPU time was spent in the FSM per-token scanner, largely in the `â` which invokes the actions for each state change. Function arrays [Be84] could reduce this overhead to a manageable figure.

# 9 Run-time Performance

Performance of the compiled code varies, as is traditional in the benchmark world, according to the application. On a SUN 386i workstation, the code runs two to five times faster than SAX (SHARP APL Unix) on the same machine. The factor of two applies to numerically intensive problems; the factor of five to highly iterative, scalar-oriented problems such as simple loops and Ackerman's function.

On the Cray X-MP, we initially were running PCC 3.0, which did not vectorize anything. Seismic applications ran roughly 800 times slower than hand-optimized FORTRAN. Although we were expecting problems in competing with FORTRAN, we were not exactly prepared for this sort of speed ratio. Luckily, PCC 4.0 did vectorize some relatively common constructs, and we boosted our performance into the ten to one hundred times slower arena.

Next, we looked to see where we were spending time in execution. We determined that calls to the run-time library were not significant, consuming less than one percent of the CPU time used. Several areas came to light as needing work, among them storage management and the C library functions.

The X-MP main storage appears to be quite slow compared to the speed of register operations. We found that the time spent in copying arguments to verbs was excessive, so we redesigned the storage manager to minimize the amount of data sloshing, by introducing reference counts.

We also wrote our own vectorized floor and square root functions, as the C run-time library only included non-vectorized, scalar versions of these functions. Finally, we ensured that our run-time library inner loops were as well vectorized as possible.

These changes got us to within a factor of two of hand-optimized FORTRAN for a convolution function. Since the APL convolution was written as a reduction of an outer product, and the FORTRAN version as an inner product, this factor of two makes sense. The project was terminated before we were able to time an APL convolution written using inner product.

The seismic Normal Move-out benchmark was still much slower than FORTRAN. We noted that most of the CPU time was spent in a verb which indexed one element out of each row of an array. This verb was written in traditional APL, using transpose and base value to create indices into the ravel of the array. Again, because of the slowness of main storage on the X-MP, the ravel was taking excessive amounts of time. I realized that the whole verb was performing the trivial SAX idiom "from rank 1" (`{ê1`), and changed the source code accordingly. This simple change sped up the entire application by a factor of 28, and got us to within a factor of six of the speed of FORTRAN. Most of the remaining time was spent within a type coercion which PCC did not vectorize.

The project terminated at this point, as we felt that we had achieved our goal of proving the feasibility of compiled APL as a supercomputing tool. Recent discussions with the C compiler writers at Cray Research lend me confidence that the new Standard C Compiler (SCC), which shares a common optimizer and code generator with Cray FORTRAN, will enable ACORN to compete very favorably with FORTRAN.

# 10 Rank Adverb Importance

I have mentioned several places where the rank adverb was of value in the compiler and in applications. In addition to its value as a tool of thought, the rank adverb has practical and theoretical benefits [Be88]. In practice, it replaces expressions involving several primitives and parentheses with a single derived verb. For example, matrix-vector multiplication, normally expressed as `((Òm)Òv)«m`, becomes `v«ê1m`. This can reduce the execution time of applications by a significant amount. In this example, three verbs are replaced by one, reducing the per-verb overhead, reducing storage management costs, and removing a large amount of data-sloshing which could not be avoided in older dialects of APL.

In a theoretical sense, the rank adverb offers advantages in language design, by offering a firm theoretical basis for making design decisions about the behavior of primitive verbs and adverbs. Its extensibility to the domain of user-defined verbs takes it beyond most other language constructs. The notion of independent application of the verb to each cell of the argument offers a simple and straightforward way to describe program decomposition across multicomputers.

The synergy of the rank adverb in combination with function arrays [Be84] creates a set of capabilities which allow excellent exploitation of MIMD computer architectures – problems can be decomposed to almost any granularity of parallelism desired. This will be discussed in a future paper.

## 11 New APL Dialects

A number of new dialects of APL are beginning to emerge. Two of these are “A” and “J” [Hu90]. These dialects are noteworthy in that they have taken the concept of function rank as absolutely basic to their design philosophy. The language facilities presume function rank as axiomatic, and the primitives are designed from that basis – there is a strong emphasis on application across the first axis, since function rank allows a direct way to move application to trailing axes. The APL alphabet, long decried by APL critics, is optional or gone, replaced by ASCII digraphs. The anomalous parts of APL syntax – brackets and semicolons – are replaced by verbs adhering to a simple, rigid syntax. As a result, J’s parse table is a simple ten by four matrix. Control structures beyond the rank adverb have been introduced, and indexed assignment has been replaced by a side-effect free merge adverb. Finally, both dialects have adopted the static scoping rules used in ACORN.

It is now clear that new dialects, rather than traditional APL, should be the basis of new work in APL compilers. The new dialects are simpler to compile, and remove many of the rough edges which APL has acquired through its long history. Furthermore, the technological and psychological objections to the APL character set are neatly avoided.

## 12 ACORN on the Connection Machine

Toward the end of the ACORN project, Walter Schwarz, then a member of the Research Department of I.P. Sharp Associates, undertook to port the ACORN run-time library to the Connection Machine. His report on that activity [Sc90] is contained in these proceedings. Suffice it to say that programming the Connection Machine for maximum performance is a non-trivial task, with extreme sensitivity to geometry and interprocess communication.

## 13 Open Questions

A number of questions remain in my mind with regard to compiled APL. What are the tradeoffs of data flow analysis versus hard declarations? Should overflow among data (Boolean to integer, integer to float, float to complex) be supported? If so, how? Is C an adequate target language, or should compilation be tied directly to an optimizer and code generator back end? Is it reasonable to share a back end with scalar-oriented languages, or will the valuable parallelism inherent in APL be lost? How should I/O and interlanguage calls be handled? Can function arrays meet all the needs of control which are otherwise missing from APL?

I am in the process of examining these questions.

## References


[Be84] Bernecky, R., "Function Arrays", *APL84 Conference Proceedings*, APL Quote Quad, Vol. 14, No. 4, 1984.

[Be88] Bernecky, R., "An Introduction to Function Rank", *APL88 Conference Proceedings*, APL Quote Quad, Vol. 18, No. 2, 1987.

[Be90a] Bernecky, R., "Ergonomics and Language Design." In press.

[Be90b] Bernecky, R., "Fortran 90 Arrays." In press, SIGPLAN Notices, January 1991.

[Bu88] Budd, T.A, *An APL Compiler*, Springer-Verlag, 1988.

[Ch86] Ching, W., "An APL/370 Compiler and Some Performance Comparisons with APL Interpreter and FORTRAN", *APL86 Conference Proceedings*, APL Quote Quad, Vol. 16, No. 4, 1986.

[Cr84] Crouch, S., *An APL Compiler*, I.P. Sharp Associates internal report, 1984.

[Dr86] Driscoll, G.C., and D.L. Orth, "Compiling APL: The Yorktown APL Translator", *IBM Journal of Research and Development*, Vol. 30, No. 6, 1986.

[Dr87] Driscoll, G.C., and D.L. Orth, "APL Compilation: Where does the time come from?", *APL87 Conference Proceedings*, APL Quote Quad, Vol. 17, No. 4, 1987.

[Hu90] Hui, R., K.E. Iverson, E.E. McDonnell, and A.T. Whitney, "APL\?", to appear in *Proceedings of APL90*, ACM SIGAPL Quote-Quad. In press.

[IS84] International Standards Organization, *ISO8485: Standard for Programming Language APL.*

[Iv80] Iverson, K.E., "APL as a Tool of Thought", 1979 ACM Turing Award Lecture, *Communications of the ACM*, 1981.

[Mc80] McIntyre, D.B.,"APL in a Liberal Arts College", *APL Users Meeting*, I.P. Sharp Associates Limited, Toronto, 1980.

[Sc90] Schwarz, W., "Acorn Run-Time System for the CM-2", *Proceedings of the 1990 Arrays, Functional Languages, and Parallel Systems Workshop*, Kluwer Academic, Boston, 1990.

[Wi86] Wiedmann, C., "Field Results with the APL Compiler", *APL86 Conference Proceedings*, APL Quote Quad, Vol. 16, No. 4, 1986.