ACORN: APL to C on Real Numbers

Robert Bernecky
18 Fifth Street
Ward's Island
Toronto, Ontario M5J 2B9
Canada
(416) 368-6944

Charles Brenner
2300 Grand Canal
Venice, California
90291
USA
(213) 827-7009

Stephen B. Jaffe
Mobil Research and Development Corp.
Paulsboro Research Laboratory
Paulsboro, New Jersey 08066
USA
(609) 423-1040

George P. Moeckel
Mobil Research and Development Corp.
Dallas Research Laboratory
13777 Midway Road
Dallas, Texas 75234
USA
(214) 851-8702

ABSTRACT

A prototype APL to C compiler (*ACORN*: APL to C On Real Numbers) was produced while investigating improved tools for solving numerically intensive problems on supercomputers. *ACORN* currently produces code which runs slower than hand-coded Cray FORTRAN, but we have identified the major performance bottlenecks, and believe we know how to remove them. Although created in a short time on a limited budget, and intended only as a proof of the feasibility of compiling APL for numerically intensive environments, *ACORN* has shown that straightforward compiled APL will be able to compete with hand-optimized FORTRAN in many common supercomputer applications.

BACKGROUND

Supercomputers are an expensive resource: they are costly, and require highly trained expert programmers to make effective use of them. In a research environment, this can be deadly – today, researchers with expertise in disciplines such as geophysics are dependent on those experts to solve their problems. In many environments, this can lead to bottlenecks and delays – a researcher wishing to model behavior of some system may have to wait months before an expert is available; alternately, he or she may be forced to write a model in ignorance, perhaps suffering one or more orders of magnitude performance degradation. When studying large problems, whose solutions may take at best hours, the spectre of days or weeks of processor time is daunting.

A language such as APL offers a possible solution. APL is an abstract language, in which you describe what you want done, not how to do it. The "how" decision is left to the computer or compiler writer. For example, the sum of a list of numbers, n, is written in APL as $+/n$, whereas scalar-oriented languages require the programmer to write a loop. As well, the performance of semantically deficient languages such as FORTRAN and C have traditionally been sensitive to the way expressions are written – interchanging two loops might make a dramatic difference in the performance of a program.

Because APL tends to bury loops within primitive expressions, loop interchange and other performance-related transformations can be made automatically and dynamically beneath the level of user visibility. This can simplify the user's program – detailed concerns about machine dependencies need not appear as explicit source code. This increases the portability of the program, when measured in terms of program efficiency on a number of machine architectures.

Architectural dependencies also affect performance, making one construct of loop faster on one system and slower on another. These dependencies hinder code portability. By hiding loops and other superfluous details, APL allows the programmer to rise above these concerns and concentrate on the essence of the problem at hand. The compiler writer can create optimal code for the specific target architecture, producing more efficient code than the average programmer is capable of providing.

The advanced semantics of APL offer another benefit. By providing primitive capabilities such as set membership, the programmer is freed from the problem of writing an efficient set membership function for a particular machine – the job is already done, and done well by a professional programmer. This may seem a trivial matter, but efficient methods for performing commonly required operations such as sorting, matrix product, and set membership differ dramatically from one machine architecture to another. Failure to describe these operations in an abstract manner cripples attempts to write portable, efficient applications.

Discussions with Stephen Jaffe and George Moeckel of the Dallas Research Laboratory of Mobil Research and Development Corporation (MRDC) led to a joint research project between MRDC and I.P. Sharp Associates Limited (IPSA), to study *the use of APL as a delivery vehicle for parallel computation.*

The basic question we have tried to answer in this project is:

Can a naive APL compiler produce code of sufficient performance and reliability to qualify it as a practical tool for solving numerically intensive problems?

We chose APL because of its outstanding track record as a prototyping and modelling tool [Be86, Yo86], and its effectiveness in conceptualizing and formulating logic. In addition, APL's advanced semantics allow it to take advantage of state of the art SIMD and MIMD computers now appearing on the scene, without requiring programmers to change their programming styles.

We felt that a naive APL compiler that performed little or no classical optimizations might, because of the powerful semantics of APL, provide adequate performance for a number of applications. There were two ways in which we thought this might occur; these are outlined below.

- A compiler performs syntax analysis once, rather than continuously, as an APL interpreter must do. Since syntax analysis often represents 10-30% of the entire processor time associated with an interpreted APL application, there are gains to be made here. The applications which suffer most from syntax analyzer overhead are those which are highly iterative or recursive, and in which the arrays being processed are very small.

- Large, numerically intensive computations of the type which are common in supercomputer applications often spend a large amount of time executing what in APL are single primitives, such as inner product, outer product, and matrix inversion. A compiler which merely optimized the run-time library for these primitives would obtain many of the speedups available in a highly optimized compiler for a language with more primitive semantics, such as FORTRAN, where the code for commonly used functions such as matrix problems are spread across a large number of primitive operations.

The original project specified hand-compilation of a few seismic applications into Cray FORTRAN or assembler code. We quickly abandoned assembler as being non-portable and non-productive. When we heard about the availability of the C language on the Cray, we decided to compile to C instead.

C, often considered a generic assembler code, offers a number of desirable facilities such as portability, simplicity, and relatively good performance. In addition, C is a more functional language than FORTRAN, so the mapping from APL to C was a fairly obvious and straightforward design problem.

Early in the study, we obtained a copy of Timothy Budd's APL to C compiler [Bu83]. This university-developed compiler is available without license fee, and we thought it might be a quick solution to our problem. Jiri Dvorak, of the IPSA APL Systems Development Department, spent some time attempting to make the compiler work, but found that the compiler was not robust enough for our use, and we dropped it.

One of us (Bernecky) started to hand-code the translated APL, but the tedium of the job quickly convinced us that writing a translator was less effort. We therefore decided to write our own compiler. Bernecky and Charles Brenner (an independent consultant under contract to IPSA) then developed a prototype compiler, dubbed *ACORN: APL to C on Real Numbers*, which we feel demonstrates both the feasibility of, and problems associated with, compiling a subset of ISO APL [ISO84] for supercomputers. Bernecky wrote the tokenizer, syntax analyzer, and code generator in SHARP APL/PC. Brenner wrote the initial run-time library in C. Jaffe and Moeckel provided a suite of seismic applications, written in APL, which served as our benchmarks.

OTHER APL COMPILERS

A number of attempts have been made to produce APL compilers. Stephen Crouch, of the I.P. Sharp Network Development Department, developed a compiler [Cr84] in 1981-1984 with restrictions similar to those of *ACORN*, which was used to produce code for the Computer Automation Alpha/LSI minicomputers then in use as IPSANET node computers.

The Budd and Sofremi [Gu85] compilers generate C code as their output. STSC's APL compiler [Wi79] compiles APL to 370 machine code. Two IBM compilers [Dr85, Ch88] are research projects; however, detailed information about their performance or internals, beyond that described in the above-cited papers, is currently unavailable. The Driscoll and Orth compiler generates FORTRAN as its output; Ching's compiler generates 370 machine code.

One underlying assumption is that these problems are computationally dominated by non-linear computations such as matrix divide, inner product, and outer product on large arrays. Such problems lend themselves well to APL, and a compiler that doesn't perform classical optimizations should perform at an adequate level. Of course, an optimizing APL compiler (*OAK*?) might make things even better, but that was beyond the scope of this research project.

COMPILER OVERVIEW

ACORN's input is the canonical representation of an APL function, or the name of an APL function in the *acorn* workspace. Its character matrix result is a C function corresponding to the input function. The cover function *compile* performs the additional work required to place the resulting C source code on a DOS file, in ASCII format.

The C functions created by *ACORN* have the following characteristics.

- APL labels and constants become static constants in the C code.
- Functions called by the APL function are presumed to be C functions created by *ACORN*.
- APL locals become C locals, represented as C structures which point into the C heap where the actual array data is stored. The C structure which represents APL arrays is described in the C TYPEDEF VAR, contained in file APL.H.
- APL globals become C static globals, represented in the same way as APL locals.
- Each primitive function or user-defined function is compiled into a C function call to a run-time library function, or to a compiled user-defined function.

In the interest of simplicity, storage management is left, as much as possible, to C. The inability to compact the heap is a potential problems for certain applications, although we have not yet had any problems in this area. If such problems do arise, then a more sophisticated storage manager, which supports compaction, may be required. This might also provide improved performance over C storage management functions.

ACORN maintains a reference count and an element count ($\times/\rho\omega$) for arrays. Reference counts allow several objects to refer to the same array without physically copying the array. Element count is frequently required by APL primitives; for example, multiplication needs to know how many elements are in the arrays to be multiplied.

Compile times on a 3090 class mainframe running SHARP APL are under a second. On a PC/AT class machine, under SHARP APL/PC, compile times are roughly one minute per line of code. Because of the prototypical nature of the work, no effort whatsoever was made to improve compilation performance. However, an order of magnitude speedup is probably achievable with a day or so of work.

COMPILER INTERNALS

The compiler consists of several phases, outlined below.

- Tokenization: This determines the class of each character in the function being compiled, and produces a character matrix of the same shape as the function's display form, with a class type for each character of the function. Failures during tokenization usually indicate use of numerics in names, use of quad (⎕), or use of character constants in a function.
- Header analysis: This analyzes the function header, and produces a C function header, locals declarations, and function prolog and epilog code.
- Syntax analysis: This is performed by a reduction analyzer, implemented as a finite state machine. Each action of the analyzer is performed by a function named *fsmXY*, where *X* is the current state of the analyzer, and *Y* is the signal, or token class, of the next character on the function line being compiled.
- Label analysis: This extracts all labels defined in the function, and produces code to define the labels in the resulting C program.
- Constant analysis: This extracts all constants from the function, generates C constants for them, and replaces occurrences of all constants with identified references to those constants.
- The compiler generates code for each line of APL, rather than trying to compile the entire function body in parallel. This was done to avoid workspace full errors, as well as for simplicity.

ACORN FEATURES

ACORN offers a number of features not normally found in other compiled languages.

- Function arguments and results have no fixed limits on rank, shape, or number of elements. This preserves the generality of APL behavior, although it does have a performance impact, particularly in areas such as arithmetic on scalars.

- Because ACORN produces C source code as its output, programs written in ACORN and C can be integrated with relative ease.

ACORN RESTRICTIONS

Because the original scope of the project was to perform a feasibility study for compiled APL, funding and time were limited. Therefore, in the interest of simplicity and rapid implementation, a number of restrictions were placed on the subset of ISO APL which may be compiled.

Furthermore, the set of primitives which were implemented, and the extent to which they were implemented, were only those required for the suite of seismic applications we were using as benchmarks. Some of the primitives were only partially implemented, because the applications didn't require the full function of the primitive.

The syntax analyzer is incomplete, which means that a number of uncommon but legal APL expressions (such as $x+(y)$) cause the compiler to fail.

- The *only* data type is floating point. This eliminates the requirement to perform semantic analysis, data flow analysis, and also handily sidesteps the issue of declarations. For many numerically intensive problems, this restriction appears to be a reasonable limitation, at least initially. For workstation applications and more general applications, it is clearly a showstopper. It also causes performance problems, as noted below.
- No character constants may be used. This is a reflection of floating point being the only data type.
- System variables and system functions are not supported.
- $\square ct$ is 0.
- $\square pp$ is 5.
- $\square io$ is 1.

The tables on the next few pages summarize the facilities and limitations discussed above.

SCALAR FUNCTIONS	NOTES
B+*C*	Scalar extension is supported
B−*C*	Scalar extension is supported
B×*C*	Scalar extension is supported
B÷*C*	Scalar extension is supported Divide by zero is not supported
B|*C*	Scalar extension is supported Doesn't comply with ISO APL for negative numbers
B○*C*	Scalar extension is supported Left argument must be 4
B⌈*C*	Scalar extension is supported
B⌊*C*	Scalar extension is supported
B⋆*C*	Scalar extension is supported
B=*C*	Scalar extension is supported ⎕ct assumed zero
B≠*C*	Scalar extension is supported ⎕ct assumed zero
B<*C*	Scalar extension is supported ⎕ct assumed zero
B≤*C*	Scalar extension is supported ⎕ct assumed zero
B≥*C*	Scalar extension is supported ⎕ct assumed zero
B>*C*	Scalar extension is supported ⎕ct assumed zero
B⌊*C*	Scalar extension is supported
−*C*	
⌊*C*	⌊¯1.2 is ¯1, not ¯2 as in ISO APL ⎕ct assumed zero
⌈*C*	Correct results, but slower than floor ⎕ct assumed zero
α,[⎕io]ω	Bracket axis operator not supported for any function

MIXED and DERIVED FUNCTIONS	NOTES
`B+.×C`	No other inner products supported No scalar extension
`B∘.×C`	
`B∘.+C`	No other outer products supported
`⌈/C`	Last axis only
`+⌿C`	First axis only No other reductions supported
`⍳C`	No error checking
`,C`	
`B,C`	No scalar extension
`B/C`	Scalars and vectors only No error checking. Scalar extension supported
`⌽C`	Last axis only
`⊖C`	Rank 2 only
`B⌽C`	Last axis only No error checking. No scalar extension
`B↓C`	No error checking. Maximum rank 2
`B↑C`	No error checking. Maximum rank 2 Error on attempted negative overtake
`B[C]`	B rank 2 or less. Elided argument supported Subscript may be of arbtrary rank Indexed assignment supported
`B←C`	Must be first function in line `a←b←c` is forbidden `→(0≠i←i-1)⍴⍳i` is forbidden
`⍕C`	Maximum rank 2 or less Print value on STDOUT Abbreviates with "..." for large array
`B⍕C`	Print entire array. Left argument ignored
`vecin k`	Get `k` element vector from input file "APLIN" File assumed to be ASCII string of numbers, delimited by one or more blanks

FINDINGS

This section presents the most notable findings of the project.

- The SHARP APL rank adverb (⍤) and from verb (⌷) proved to be effective tools in circumventing some performance problems present in ISO APL. In particular, the NMO benchmark originally used a defined function, *index*, to perform scatter indexing of points from a matrix. ISO APL is not very effective at such scatter indexing, and a large amount of the NMO CPU time was spent in *index*. Rewriting the *index* function as the SHARP APL expression `x⌷⍤1 y` made a dramatic improvement in performance. Additional improvements could be made in NMO by using the rank adverb with multiplication and catenation. These changes would simplify the APL code as well as reduce the amount of superfluous data movement required.

- Use of static, rather than dynamic scoping, dramatically simplified the problem, without severely impacting practical applications. It allowed independent compilation of each function, without requiring knowledge of the calling tree.

- Although the Cray hardware supports compress and compress-iota, Cray C does not appear to provide any way of accessing those facilities. It is possible that performance gains could be made by writing Cray assembler code routines to support these functions, once *ACORN* supports Boolean data as bits. Furthermore, in contrast to languages such as FORTRAN, APL's powerful semantics open an effective window into the Cray hardware, without impacting application portability.

- Use of reference count techniques for array storage management produced significant performance improvements, and reduced storage requirements dramatically. These effects are observable in interpreted APL, but are of greater importance on the Cray XMP, because of the Cray's relatively slow main storage access times.

- Use of the compiler across three machines is a clerical nuisance: The compiler runs on a PC/AT. The C source file it produces has to be uploaded from the PC/AT to the VAX front-end, and then transferred to the Cray for compilation and execution. Cray job control and link-edit statements for compiling and linking the C programs on the Cray must be manually maintained. If the APL functions to be compiled reside on the VAX, they must be downloaded to the PC/AT. A seamless, pleasant ("screamless") development environment is a priority item for effective use of this technology. Adapting the compiler to run on the VAX front end would be a considerable improvement.

- Treatment of the niladic "main" function required by C programs is somewhat clumsy, because of C's inability to accept a direct argument to the "main' program.

- Since ACORN produces code which consists almost entirely of function calls to the run-time library, we were initially concerned about the overhead of C function calls. We found that on the Cray, function call overhead was small, less than one percent of the entire benchmark time, and hence was not a serious problem.

- Another concern was whether or not C storage management functions would perform well enough to let us avoid having to develop a sophisticated storage manager based on storage pooling. Our chief worry was that storage fragmentation would cause allocation of a large array to fail, even though such space was available, but non-contiguous. In the interest of empiricism and simplicity, we adopted C storage management functions without storage pooling. This works adequately, as we have not yet observed any storage management problems which could be attributed to fragmentation.

- Introduction of "traditional" control structures, such as DO, WHILE, IF/THEN/ELSE, would improve the quality of code which could be generated in a more sophisticated compiler. In ACORN, the main observable effect of this omission is generation of clumsier code to support branching than other control structures would require.

ACORN PERFORMANCE ON WORKSTATIONS

We initially used SUN 386i workstations and IBM PCs as development and test platforms. Several benchmarks were used to measure the performance of the resulting compiled code on the SUN against interpreted APL on the same machine. These are described below.

- BENCHLOOP: A simple, scalar-oriented loop. This is a typical example of code on which interpreted APL usually performs poorly.

- ACK: Ackerman's function, with arguments of 3 and 4. This heavily recursive function is a good measure of the performance of defined function call and scalar performance.

- NMO: This function is a seismic "normal move out" application. It is numerically intensive, using outer products, reductions and interpolations, on large arrays.

- CONV: This function performs convolution on vectors, using a reduction of an outer product between the seismic trace and the rotated filter. Although we had two versions of convolution, one using reduction of outer product, and one using inner product, we only timed the outer product version. The APL function used for convolution was:

```
∇ r←wz conv tr;npad;h
[1] h←wz∘.×tr,(npad←(⍴wz)-1)⍴0
[2] r←(⍴tr)↑+⌿(0,-⍳npad)⌽h
∇
```

The following table displays the relative performance, in CPU seconds, of some of the compiled code on a SUN 386i, running SUN OS, compared to the SAX APL interpreter on the same platform.

APPLICATION	SAX/386i	ACORN/386i
BENCHLOOP 10	56	11
3 ACK 4	34	10
NMO	33	16.5
CONV	n/a	n/a

ACORN PERFORMANCE ON CRAY XMP

ACORN performance varies depending on the host and the application. For the particular cases of CONV and NMO on the Cray, ACORN produced code which executed four and seven times slower than hand-optimized FORTRAN on the Cray XMP, respectively. In the timings below, the CONV timings are per element of the right argument, for the filter size given.

APPLICATION	Cray FORTRAN	Cray ACORN
`(102⍴filter) conv 16000⍴trace`	1.2 μsec/element	4.6 μsec/element
`nmo 60 1000⍴trace`	26 msec	187 msec

We attribute the performance difference to several factors, which are discussed in more detail below.

- Cray C compiler inadequacy: The Cray C4.0 compiler does not optimize as well as the FORTRAN compiler – a simple matrix product algorithm written in C and FORTRAN took 40% longer in C than in FORTRAN. Cray claims that the next release (C5.0) of their C compiler will share a common back-end with FORTRAN, and hence should be able to generate code which performs at least as well as FORTRAN.

- Lack of vectorized run-time library routines: The C4.0 run-time library did not contain vectorized versions of square root (required by 40 ω), floor, and residue. Cray claims that C5.0 will contain vectorized versions of these functions.

- Lack of proper data type support: ACORN supports only double-precision floating point data. This has the highly undesirable effect of requiring type coercion of all values used for indexing. Our budget and time frame did not permit us to successfully vectorize this coercion. This resulted in severe degradation of NMO performance.

- APL dialect: The APL dialect used in CONV and NMO reflects APL language design as of 1970. Since then, a number of new capabilities, such as the rank adverb, have come into the language. These capabilities allow significant reduction in program complexity, and reduce the amount of main store accesses required to perform many common operations, without sacrificing portability. However, even with these improvements, APL tends to require additional operations in order to set up arguments to allow direct use of inner product, and so on. In CONV these contributed to a 19% overhead which would not be required in FORTRAN.

- Unfamiliarity: Neither Bernecky nor Brenner had any previous experience with Cray hardware. It is likely that we were ignorant of machine characteristics which, had we exploited (or avoided) them, may have resulted in improved performance.

We believe that C compiler improvements and improved vectorization should allow ACORN to perform comparably to FORTRAN. A properly designed APL compiler, including data type support, generating C code instead of run-time library calls, should, because of APL's advanced semantics, be able to match or outperform FORTRAN in almost any application. Particularly, these performance improvements should be apparent when the application must be run on a variety of machine architectures, which would forbid architecture-specific optimizations in the FORTRAN source program.

Performance Analysis of Convolution

The performance of ACORN on convolution was about one fourth as fast as hand-optimized, unrolled FORTRAN on the Cray XMP. Although we did not analyze the performance difference in detail, we attribute much of the performance loss to extra storage operations required by ACORN, compared to FORTRAN. This is characteristic of a non-optimizing APL interpreter or compiler, and is not likely to be easily corrected in a compiler as naive as ACORN.

- The SHARP APL UNIX (SAX) tesselation adverb could be used in conjunction with other adverbs to describe the convolution algorithm in a more terse fashion as:

```
(1,⍴⍵z) 3⍤⍤(+.×⍤⍤⍵z) tr.
```

The APL Dictionary [Iv87] describes these facilities. Making effective use of this expression would require significant redesign of ACORN, but might fall out of a more sophisticated compiler.

Performance Analysis of NMO

The performance of NMO was limited by C compiler restrictions, as well as by ACORN's inability to generate integer data types. In ACORN, NMO ran about one seventh the speed of hand-coded FORTRAN.

C4.0 run-time library's lack of vectorized functions for square root, coercion, floor, and residue caused this performance problem, but we believe that the next release of the Cray C compiler will resolve this.

SUMMARY

We believe that a production-quality APL compiler will:

- provide researchers with a better tool of thought
- perform as well as FORTRAN and other more traditional languages
- provide researchers with a programming tool which is portable without *any* changes among disparate machine architectures
- allow researchers and software developers to develop applications and models for supercomputers in a far shorter time frame than is possible with more primitive languages. This time advantage offers a key market edge to those who are designing new products with the aid of supercomputers.

These characteristics will combine synergistically to allow researchers whose major discipline may not be computing to make far more effective use of computers and their own time than they are able to do today.

ACORN has opened the door into a realm of more intelligent and effective use of supercomputers and workstations, and has planted the seeds of further development in that fertile area. Our next task is to enter that realm, and cultivate its land, so that we may reap the benefits of deeper understanding of our world and universe.

ACKNOWLEDGMENTS

We received considerable assistance in the use of SUN workstations and UNIX from Mark Czerwinski, Walter Schwarz, and Heather Bowen. Gordon Ross provided valuable assistance in the installation and use of IBM/370 C compilers. Don Isgitt and Dale Mihalyi assisted us in the generation of seismic test data and educated us in the use of the Cray. Elena Anzalone edited the report, improving its readability and organization. Any problems with layout, formatting, and content are Bernecky's doing.

BIBLIOGRAPHY

This bibliography contains a number of entries for documents which are not cited in the report, but which are relevant to the problem.

Ab70 Abrams, P., An APL Machine. SLAC report #114, Stanford University, 1970.

Be86 Bernecky, R., APL: A Prototyping Language. APL86 Conference Proceedings, 1986.

Bu81 Budd, T. and Treat, J., Extensions to Grid Selector Composition. University of Arizona Technical Report 81-17, 1981.

Bu83 Budd, T. A., An APL Compiler for the UNIX Timesharing System. APL83 Conference Proceedings, 1983.

Ch88 Ching, W. and Xu, A., A Vector Back End of the APL370 Compiler on IBM3090 and Some Performance Comparisons. APL88 Conference Proceedings, 1988.

Cr84 Crouch, S., Real-time APL Compiler, Version 0.0. I.P. Sharp Associates Internal Document, 1984.

Dr85 Driscoll, G.C. and Orth, D.L., APL - Compilation - Where does the time come from? APL85 Conference Proceedings, 1985.

Gu78 Guibas, L. and Wyatt, D., Compilation and Delayed Evaluation in APL. Conference Record of the Fifth ACM Symposium on Principles of Programming Languages, 1978.

Gu85 Guillon, A., An APL Compiler: The SOFREMI-AGL Compiler, A Tool to Produce Low- cost Efficient Software. APL85 Conference Proceedings, 1985.

ISO84 ISO8485: Standard for Programming Language APL.

Iv87 Iverson, K.E., *A Dictionary of APL*, APL Quote Quad, Vol. 18, No. 1, September 1989.

Ka78 Kaplan, M. and Ullman, J.D., A General Scheme for the Automatic Inference of Variable Types. Conference Record of the Fifth ACM Symposium on the Principles of Programming Languages, 1978.

Ko85 Koster, A., Compiling APL for Parallel Execution on an FFP Machine. APL85 Conference Proceedings, 1985. *This paper contains a fairly extensive bibliography on other relevant publications.*

Mi79 Miller, T., Type Checking in an Imprefect World. Conference Proceedings of the Sixth ACM Symposium on the Principles of Programming Languages, 1979.

Mu81 Muchnick, S.S. and Jones, N.D. (eds), Program Flow Analysis: Theory and Applications. Prentice Hall, 1981.

Ru85 Rudd, J. and Klementis, E.M., APL to ADA Translator. APL85 Conference Proceedings, 1985.

St77 Strawn, G.O., Does APL Really Need Run-time Parsing. Software – Practice and Experience, 7:193-200, 1977.

We81 Weiss, Z., and Saal, H.J., Compile Time Syntax Analysis of APL Programs. APL81 Conference Proceedings, 1981.

We85 Weigang, J., An Introduction to STSC's APL Compiler. APL85 Conference Proceedings, 1985.

Wi79 Wiedman, C., Steps Toward an APL Compiler. APL79 Conference Proceedings, 1979.

Yo86 Yoshino, M., APL as a Prototyping Language: Case Study of Compiler Development Project. APL86 Conference Proceedings, 1986.